UNCLASSIFIED

IDA PAPER P-1900

TOWARD Ada* VERIFICATION: A COLLECTION OF RELEVANT TOPICS

Compiled and Edited by

Richard Platek
David Gauspari
Jeffrey Hird
Doug Webber

Additional Contributions by

John McHugh
Karl Nyberg
Raymond J. Hookway

June 1986

Prepared for
Ada Joint Program Office (AJPO)

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

UNCLASSIFIED

IDA Log No. HQ 85-30670

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
June 1986

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Toward Ada Verification: A Collection of Relevant Topics

5. FUNDING NUMBERS
MDA 903 84 C 0031
T-D5-304

6. AUTHOR(S)
Richard Platek, Jeffrey Hird, David Gauspari, Doug Webber, John McHugh, Karl Nyberg, Raymond J. Hookway

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute for Defense Analyses (IDA)
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-1900

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office (AJPO)
Room 3D139, The Pentagon
Washington, D.C. 20301

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

IDA Paper P-1900, *Towards Ada Verification: A Collection of Relevant Topics*, is a survey of various topics in Ada verification, the state of progress in both near-term and far-term goals, and some possible directions for future work. Chapter 1 studies the extent to which the existing general purpose literature on program verification already covers the Ada language and concludes with an annotated bibliography. Chapter 2 surveys the possibility of adapting existing verification tools as near-term vehicles for Ada verification. Chapter 3 is a discussion of specification languages, cast in the form of a commentary on and criticism of ANNA, with suggestions for some extensions and improvements to it. Chapter 4 describes the far-term European project for a formal definition of the whole of Ada. It concludes with a brief discussion of the possibility of standards for the acceptance of verification systems. Chapter 5 presents the results of an attempt to survey as wide as possible a community of users, in particular, Ada users, on the ways in which, if at all, they use features of Ada which currently present problems for verification.

14. SUBJECT TERMS
Ada Programming Language; Verification; ANNA; DIANA; CAIS; Gypsy Verification System; SMoLCS; AVID; Modula Cornell Synthesizer-Generator.

15. NUMBER OF PAGES
138

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

IDA PAPER P-1900

TOWARD Ada* VERIFICATION:
A COLLECTION OF RELEVANT TOPICS

Compiled and Edited by

Richard Platek   David Gauspari
Jeffrey Hird   Doug Webber

Additional Contributions by

John McHugh
Karl Nyberg
Raymond J. Hookway

June 1986

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-D5-304

Table of Contents

Appendix A

Preface

This Memorandum Report was completed by Odyssey Research Associates for the Institute for Defense Analyses (IDA) and is a survey of various topics in Ada verification, the state of progress on both near-term and far-term goals, and some possible directions for future work. Additional contributions were made by John McHugh, Carl Nyberg, and Raymond Hookway; these contributions appear in Appendix A.

Chapter 1 studies the extent to which the existing general-purpose literature on program verification already covers the Ada language. It begins with a list of specific rules for constraining Ada programs so as to make the existing theory of "Floyd-Hoare style" verification applicable to them. Other sections discuss the reasons for the rules and the possibility of devising co-ordinate restrictions on compilers. One might undertake to verify certain programs only with respect to some class of legal compilers which are, in specified ways, more deterministic than the language manual strictly requires. The chapter concludes with an annotated bibliography.

Chapter 2 surveys the possibility of adapting existing verification tools as near-term vehicles for Ada verification. This chapter also considers a second near-term question: whether such emerging standards such as DIANA and Common Ada Programming Support Environment (APSE) Interface Set (CAIS) will have any impact on the design of verification tools.

Chapter 3 is a discussion of specification languages, cast in the form of a commentary on and criticism of ANNA, with suggestions for some extensions and improvements to it.

Chapter 4 describes the far-term European project for a formal definition of the whole of Ada. To our knowledge this is the only such project currently under way. In the long term-any Ada verification system will have to be based on a standard formal semantics. The chapter concludes with a brief discussion of the possibility of standards for the acceptance of verification systems.

Chapter 5 presents the results of an attempt to survey as wide as possible a community of users (in particular, Ada users) on the ways in which, if at all, they use features of Ada which currently present problems for verification. It is hoped that researchers will find in it some guidance as to which of the currently mysterious parts of Ada most merit study.

1. TOWARD ADA VERIFICATION

1.1. Introduction

1.1.1. Outline

This chapter is an attempt to discover and describe, as precisely as possible, restrictions on the writing of Ada programs which will forbid the use of constructs or combinations of constructs which are clearly beyond the capacity of current methods of program verification. It incorporates criticisms made at the workshop on Ada verification held at IDA, March 18 - March 20, 1985.

It is organized as follows: Chapter 2, following the order of LRM (the Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983) is a list of rules for programmers. Checking whether the rules have been followed is straightforward and mechanical, and requires no reference to the dynamic behavior of programs (from which it follows that they are often highly, and unnecessarily, restrictive).

For anyone familiar with the literature of program verification, Chapter 2 will contain no surprises. It is an account of how to incorporate into Ada the "classical" restrictions that are currently imposed on languages designed with verification in mind, and forbids those constructs for which no substantial principles of verification are known. "Known" means are those discoverable by a survey of the current (and applicable) literature. The expert will see, for example, that our restricted subset essentially limits Ada tasking to the resources of CSP [HOARE 78].

Chapter 2 also summarizes, informally, the kind of information a programmer would have to make available (e.g., "loop invariants") to a would-be verifier. We have attempted to make this section a practical document usable by working programmers. Accordingly, we emphasize formulations that are clear and easily understood, even if they therefore become more severe than strictly necessary. Detailed justifications of the rules, if provided at all, will be deferred to later sections, which also describe ways in which some of the restrictions may be relaxed.

Chapter 3 is written for the non-expert in verification or in Ada. It contains, for example, definitions of "aliasing" and "side-effects", reviews the terminology of access variables and access types, quotes the relevant parts of LRM's account of undefined variables, etc. Most discussion of the idiosyncracies of Ada is deferred to Section 4.0.

Chapter 4 contains a discussion of program errors—"erroneous programs" and "incorrect order dependences". Programs which are "erroneous" are sensitive to semantical decisions which have deliberately been left undetermined by Ada's designers. As different implementations will settle them in different ways, the behavior of such programs will be implementation dependent. Strictly speaking, the semantics of such programs is undefined.

Chapter 5 is an annotated bibliography. It cites, in addition to the Ada literature, several of the standard papers in program verification which were found useful in preparing this survey.

The paper concludes with an index, and we conclude this introduction with some theoretical questions about the limitations of this paper and possible directions for further work.

1.1.2. Limitations

By a "verification" we mean a correctness proof which is, if not fully automated, at least machine checkable. We have limited ourselves to considering proof techniques currently available in the literature. The commonest are the logical calculi of "Hoare-triples", assertions of the form "If condition A holds when the execution of program P is initiated then—subject, perhaps, to additional hypothesis—condition B will result." The most common additional hypothesis is the hypothesis that the program P terminates—that is, that B will hold upon termination of P, provided that P does indeed terminate. Our first approximation added another: that no predefined exceptions be raised during execution of P. This has been criticized as being unnecessarily restrictive. The opinion and the criticisms are further explained in the discussions of exception handling.

We do not claim to have a model of the "allowed" portion of the language; or that there are usable proof rules for arbitrary combinations of the "allowed" constructs; or, a fortiori to guarantee that any program written with the "allowed" constructs and accompanied by the appropriate sorts of comments can indeed be verified from the rules in the literature. All we can say is that programs which violate restrictions lie comfortably within the large domain of current ignorance.

The hypothesis that P terminate is the most common one to make in axiomatizing sequential programming languages—or, to be more precise, in axiomatizing that part of programming which consists in the computation of functions. But there is an important distinction between constructs or modules which are intended to terminate and those which are not. If a module is intended to terminate, then its effect is reasonably describable by Hoare triples as an input-output relationship, and failure to terminate is simply a mistake. A program unit which is not intended to terminate ordinarily provides a service—for the moment we'll call them "services". The design of an appropriate language for speaking about services is a subject of active research. One Hoare-like strategy for specifying a service is to record an invariant which holds true either at every moment of the service's life, or at all moments except those explicitly bracketed off. If this strategy is adopted it may then seem unreasonable not to follow the strategy completely—making the whole logic a logic of invariance. This is also a subject of active research. We mention the problems posed by non-terminating program units simply to take note of a difficulty with the approaches surveyed in this paper.

One final question, which might well have come first: just what are we verifying—the logic of the source text or the behavior of the compiled code? The problem is not the possibility of bugs in the compiler but the variations in behavior that can result from optimizing compilers acting quite legally. The discussion in Chapters 2 through 4 of initialization and undefined variables presents an example of the difficulty.

1.1.3. One, Two, Many Systems: Tractable "Clusters"

We think it would be useful to mark out several "verifiable" sublanguages of Ada. The main reason for this is well-illustrated in the literature. Imagine a language with the constructs R, S, and T—and suppose that one has in hand a usable proof system for programs in which only R and S occur. It's quite possible that incorporating T into the proof system would require not only the addition of rules for T, but also complications in the rules for R and S – so that even the proofs for programs involving only R and S become more difficult. For example, introducing aggregate types like records and arrays complicates the logic of procedure calls. More generally, handling the logic of procedure calls requires a more detailed than usual analysis of the assignment statement.

It may pay to set aside the subset {R,S} as a "verifiable" subset with its own simple proof system. For example, time-critical or space-critical applications are unlikely to use recursive subprograms. It's at least thinkable that one could verify programs using large amounts of the language if each program unit were restricted to some tractable combination of constructs (thereby hiding the awkward combinations from one another).

A fancy way to say this is to say that one might hope to devise a proof system which is not context free. Many of the problem features of Ada are well understood: one understands them well enough to be confident that in their full generality they are intractible to verification. By restricting the contexts in which they occur or the range of options open to compilers (see next section) it may be possible to domesticate them.

We note, finally, that correctness is not the only goal of software engineering, and it might also be useful to carve out a variety of "verifiable" subsets corresponding to a variety of other goals such as modifiability and portability. All these concerns, however, often point in the same direction.

Among the advantages of modularizing the proof system: it immediately suggests some concrete things to do, namely, to study the requirements of individual problem domains and look for useful tractable subsets. It doesn't prejudge any questions of technique—nothing requires that different subsets be attacked by the same methods, or by methods that can easily be integrated with one another. Finally, the syst:m can be improved piecemeal. One can introduce new subsets at will, or incorporate an additional construct into an existing subset without having to incorporate that construct into any others.

1.1.4. Predictable compilers

The concerns of compilers and verifiers overlap. Compilers may use the results of verification to help optimize their performance (for example, by suppressing certain run-time checks shown to be unnecessary). A verifier may verify a program on the assumption that all aggregates are passed by reference, or that certain (optimizing) re-orderings of computations will not be attempted, or that certain run-time checks will always be made.

The definition of Ada leaves many semantically consequential details to the discretion of its implementors. From the standpoint of verification, this discretion is sometimes too broad. We therefore begin to explore the possibility of verifying programs relative to general classes of compilers. A relative verification would contain the proviso: so long as the program has been compiled on a "predictable" compiler" or "on a compiler predictable in such and such ways." This possibility has been raised, independently, by N. Cohen, who suggests the term "natural" compiler.

Notice that restrictions on compilers complement restrictions on the language – roughly, more restrictions on compilers make larger clusters possible. Predictability might be implemented by some combination of outright limitations on the discretion of compilers and pragmas which could be invoked to impose these restrictions selectively.

Some simple illustrations are provided in Chapter 4.

1.2. Rules

All references to sections and chapters of the Reference Manual for the Ada Programming Language are preceded by "LRM", e.g., "LRM 3.2.1."

LRM Chapter 1, Introduction

No restrictions.

LRM Chapter 2, Lexical Elements

No restrictions.

LRM Chapter 3, Declarations and Types

LRM 3.1 - LRM 3.2

No restrictions.

LRM 3.2.1 Object Declarations

If a variable is not initialized at the time of its declaration, there is no convenient set of syntactical rules to ensure that it will ever receive a value. In particular, exceptions may be raised between declaration and initializatio, even if, in the program text, no executable statement separates declaration from initialization (see Chapter 4). A program which attempts to evaluate a scalar variable whose value is undefined or attempts to apply a predefined operator to a variable any of whose scalar subcomponents is undefined is erroneous (LRM 3.2.1, 6.2) and its effect is therefore unpredictable. Below we set out a rather severe discipline which navigates around almost all of these difficulties. Some of these are so awkward that they might be regarded as a proof that compiler restrictions are necessary in order to avoid them. Ways of mitigating this severity are discussed in Chapter 4.

It should also be noted that in some cases [LEDGARD 82] suggest quite the opposite of what we suggest here (see Chapter 3).

- A declaration of a record type may, and therefore must, provide default values for variables of that type (LRM 3.7).
- Access variables automatically have the default value null and therefore need no explicit initialization.

- The evaluation of an allocator may and therefore must initialize the object designated by the access variable being allocated. Exception: Suppose T is a task type, type POINT is access T, and t is a variable of type T. Then execution of "t := new T" defines the value of t (LRM 9.2). Explicit initialization is neither necessary nor possible. (Note: Later restrictions will rule out pointers to task types, making this exception moot.)

- Variables of limited private type (other than task types) must be implemented as records, so that they may be given default values.

- Variables of all other types may, and therefore must, be initialized upon declaration, with the following exception: variables may be attached (via address clauses) to addresses which are hardware-controlled and these cannot be initialized by program declarations. Further, certain addresses may have special significance for the operating system and need not be initialized by program declarations. The compiler must know which addresses are "wired"—so that it will not raise program_error on the grounds that variables assigned to those addresses are not undefined.]

- If a variable is declared in the visible part of a package P (and therefore initialized by the declaration) and it or any of its subcomponents is potentially altered by execution of the package body, then any context clause which names P must be followed by the pragma ELABORATE(P). In this way, any to allowed sequences of elaborations will have the same effect. A program sensitive to the order in which program units are elaborated contains an incorrect order dependence (LRM 1.6, 10.5; see also Chapter 4).

- There is one gap in our set of rules: the formal out parameters of a procedure. "The value of a scalar [out] parameter that is not updated by the call is undefined upon return; the same holds for the value of a scalar subcomponent, other than a discriminant." (LRM 6.2) Note: this is true even if the formal parameter is of a type having a default value. The rules described above will not in general be sufficient to guarantee that out parameters always receive values, precisely because there is an unavoidable gap between declaration and initialization (see Chapter 4).

LRM 3.2.2

No restrictions.

LRM 3.3 Types and Subtypes

Task types may be used only as templates – that is, access types to task types will be forbidden. (See restrictions to LRM 3.8 and 9.)

LRM 3.4 - LRM 3.5.6

No restrictions.

Note: LRM 3.5.4 says that an integer type with range L..R is, semantically, a subtype of an anonymous, implementation-dependent type whose range of values is at least L..R.. This suggests compiler restrictions which would make the relation between an integer type and its anonymous base type logically clean. We have chosen this as one of our examples illustrating the nature of a predictable compiler (see Chapter 4).

ARM 3.5.7, 3.5.8 Floating Point Types, Operations

The difficulties here are well known and are not of the kind that would be solved by restricting the language. Some theoretical remarks are included in Chapter 4.0.

LRM 3.5.9 - 3.8

No restrictions.

LRM 3.8 Access Types

Access types to task types are forbidden.

LRM 3.9

No restrictions.

LRM Chapter 4, Names and Expressions

LRM 4.1 - 4.1.3

No restrictions.

LRM 4.1.4 Attributes

We simply note that the "logic" of the attributes has not been studied, and should be. The dynamic attributes of tasks (such as TERMINATED and CALLABLE) are known to present theoretical difficulties. On prudential grounds then, we'll forbid the use of attributes.

LRM 4.2 - 4.7

No restrictions.

LRM 4.8 Allocators

The evaluation of an allocator may and therefore must, according to the rules given above, initialize the object designated by the access variable being allocated. As noted above, objects of task type which are created by the evaluation of allocators are automatically defined.

LRM 4.9 - 4.10

No restrictions.

LRM Chapter 5 Statements

LRM 5.1 - 5.4

No restrictions.

LRM 5.5 Loop Statments

The classical requirement for Hoare-style calculi is that the programmer supply an invariant true at each re-entry of the loop and an invariant true at each potential "exit" or "return" statement. As noted in the introduction, this does not suffice to specify loops which are intentionally non-terminating. Loops can also be quit by the raising of exceptions. Using an error as the normal exit from a loop is usually regarded as poor practice, however [LUCKHAM 80] say that they have not found such a use of exceptions burdensome.

LRM 5.6 - 5.8

No restrictions.

LRM 5.9 Goto Statements

Goto statements are forbidden — a restriction more a convenience than a necessity.

LRM Chapter 6 Subprograms

LRM 6.1 Subprogram Declarations

The body of a subprogram may not contain a declaration for another subprogram of the same name and parameter type profile. ("Parameter type profile" is defined in LRM 6.6).

Tasks or objects with tasks as subcomponents may not be passed as parameters to subprograms. (This is a consequence of the broad ignorance of tasking.)

A subprogram must be accompanied by a list of the global variables occuring in the subprogram. Note that the set of global variables occurring in subprogram P must be defined recursively. It consists of those variables which are governed by declarations occurring outside the scope of P and which either are global variables of subprograms called on by p or are variables occurring in the body of P. See the discussions of LRM 6.4 and LRM 6.5.

LRM 6.2 Formal Parameter Modes

As noted in the discussion of LRM 3.2.1, the following discipline does not quite avoid all the obstacles to initializing variables: The formal out parameter of a procedure must be initialized at the beginning of the procedure body (see Chapter 4).

LRM 6.3 Subprogram Bodies

See 6.1, 6.2.

LRM 6.4 Subprogram Calls

Refer to the comments about LRM 6.1 - LRM 6.3.

We impose a stringent "no aliasing" rule. Chapter 3 elaborately defines the phrase "such and such variables are aliased" and the broader notion "such and such variables have related names." Because one can determine at compile time whether two variables have related names (and cannot always determine at compile time whether they will be aliased), we formulate our rule as follows:

- No two actual parameters to a subprogram may have related names unless both are in parameters.
- No actual parameter to a subprogram may have a name related to a global variable of that subprogram. (Note: if the subprogram is declared within the scope of the declaration of one or more access types, the notion "global variable" is somewhat subtle. See Chapter 3)
- If any identifiers occurring in the actual parameters or global variables have been introduced by renaming declarations, the test for "related names" must be applied after the renamings have been eliminated.

Ways to mitigate the severity of this rule are discussed in later sections.

Warning to experts: the typical "no aliasing" restrictions in Pascal-like languages allow var parameters to be aliased against val parameters, so long as the specification describing the effect of the procedure body is of a certain constrained logical form. The Ada parameter mode in does not correspond fully to val, and the Ada modes in out and out do not correspond fully to var – because the Ada modes do not determine the method of parameter passing. That is why we rule out any aliasing of in against in out or out. A fuller account of the effects of Ada's underdetermined semantics will be deferred to in Chapter 4 (erroneous programs).

LRM 6.5 Function Subprograms

- Preferred usage: global variables should not occur in function bodies. If globals do occur, their names must not be "related to" those of any actual parameters. (For the definition of "related names", see Chapter 3.)

- Functions must not have side-effects. This means that a function body may not contain the following:

- Assignments to global variables or calls to procedures which can change global variables

- I/O operations

- Allocators: Statements of the form "x := new T", where x is an access variable designating objects of type T

- Occurrences of "run-time" attributes: Attributes whose values can change during execution (this seconds the general restrictions which will be imposed below on the use of attributes)

- Assignments to subcomponents of access variables, or calls to procedures which make such assignments

These matters are further discussed in the next section.

LRM Chapter 7, Packages

We repeat the rule laid down in the discussion of LRM 3.2.1: if a variable is declared in the visible part of a package P (and therefore initialized by the declaration) and it or any of its subcomponents is potentially altered by execution of the package body, then any context clause which names P must be followed by the pragma ELABORATE(P).

Except for its effects on the variables declared in the package specification, execution of the package body should have no side-effects on entities visible outside the package body. That is, treat the package body as a function body and apply the "no side-effects" rules given in the discussion of LRM 6.5.

LRM Chapter 8, Visibility Rules

No restrictions

LRM Chapter 9, Tasks

What's known about tasking is rather limited, and the rules below are no more than an indication of the kinds of limitations that have so far been imposed to isolate tractable fragments of the language. The bibliography in Chapter 5 sets out in detail the fragment of

Ada tasking treated in each of the included papers.

- The collection of tasks must be fixed, the tasks must begin together, and the tasks must themselves be sequential, i.e., they must not create further tasks by declaration or allocation. Accordingly, a task may not be declared within a task and there can be no access types to task types.

- Tasks may not share memory. Accordingly, tasks may not have global variables in common and may not pass access variables in a rendezvous. Warning: any occurrence of shared memory is unsafe, even if the shared variables can be updated only by by passing them as in out parameters to a third task (see Chapter 4).

- Entry calls must obey all the restrictions (against aliasing) imposed on procedure calls.

- The attributes COUNT, CALLABLE, or TERMINATED may not be used. This restriction effectively prevents the programmer from writing his own scheduler.

- The abort statement may not be used.

- delay statements may not be used. As a general rule, the logic of the real-time features is not understood.

LRM Chapter 10, Program Structure and Compilation Issues

See the requirement on the ELABORATE pragma in the discussion of LRM 7 or LRM 3.2.1. Section 4.2 of this document discusses incorrect order dependences that could arise among separately compiled program units.

LRM Chapter 11, Exceptions

The rules which we set out here are a makeshift, short on both theoretical understanding and practical experience with Ada programs. The rules will simplify the verifier's task, but may not make it simple enough (see Chapter 3). They use the exception mechanisms of Ada to mimic something approximating those of Gypsy [GOOD 84]. However, Ada is intrinsically more complex than Gypsy and these differences can't be fully papered over.

The basic principles are as follows: subprograms must not be exited by means of unhandled exceptions. Exceptions must not be propagated beyond their scopes. (Note: There is no syntactical guarantee that the first of these conditions will be met, since unhandleable exceptions can occur during execution of the body of the handler.)

These principles can almost be enforced as follows:

- Define a global exception ROUTINE_ERROR.
- Every subprogram must contain an exception handler with the clause: when others => raise ROUTINE_EXCEPTION.
- If the body of an exception handler contains a raise statement, the exception raised may not be local to the subprogram. In particular, a local exception may not be re-raised.

Following [LUCKHAM 80], we say that the programmer must supply for each handler an assertion true of any state which will result from execution of that handler.

LRM 11.6, Exceptions and Optimization

[COHEN 85] observes that optimizing compilers which reorganize computations present difficult problems to verifiers. Some legal reorganizations, for example, can cause errors to be raised that would not otherwise be raised or to alter the place at which an error is raised. This presents a problem for verification which can't be solved by syntactical restrictions alone.

LRM Chapter 12, Generic Units

The restrictions are imposed on the use of a generic X are those imposed on the use of X.

LRM Chapter 13, Representation Clauses and Implementation-Dependent Features

We simply offer some observations.

LRM 13.1 - 13.4, Representation Clauses

No restrictions.

LRM 13.5, Address Clauses

The effects of these address clauses are highly machine dependent. A stand-alone verifier can only do one of two things: forbid them entirely or carry out a proof modulo the assumption that they contain no mistakes. In particular, this assumption includes the following:

- That they cause no overlays, and do not link a single interrupt to more than one entry (erroneous, ARM 13.5).
- That, used in conjunction with representation clauses to establish hardware interfaces, they do it right.

LRM 13.6

No restrictions.

LRM 13.7 - 13.9 The SYSTEM Package, Machine Code Insertions, Interface to Other Languages

Use of the SYSTEM package (including representation attributes), of machine code insertions, or of interface to other languages is forbidden. The restriction on use of SYSTEM is purely prudential: without further study it's not clear what, for example, will be the logical effects of programs which refer to the finiteness of the machine on which they're being run.

LRM 13.10 Unchecked Programming

Unchecked_conversion (being completely implementation dependent) is forbidden.

In the absence of a verification that no attempts will be made to acess dangling pointers unchecked_deallocation is forbidden.

LRM Chapter 14 Input-Output

To our knowledge, little theoretical work exists on specifying and reasoning about I/O. We simply repeat the single reference to I/O which has appeared in the discussions of other constructs:

I/O operations are forbidden in function or package bodies. (See rules for LRM 6.5 and LRM 7. See also Chapters 3 and 4, below.)

LRM Appendix A, Predefined Language attributes

Both predefined and user defined attributes have been disallowed for reasons of prudence.

1.3. Discussion of the rules and alternative rules

This section discusses some of the rules set out in Chapter 2, and also notes ways in which at the expense of some complication the rules can be liberalized. Relevant sections of the LRM are quoted from or summarized. The special subject of erroneous programs and incorrect order dependencies is reserved for Chapter 4.

1.3.1.· (Initialization: LRM 3.2.1)

Here are the relevant texts on defined and undefined values:

LRM 3.2.1 Object Declaration_

- The result of an attempt to evaluate an undefined scalar variable, or to apply a predefined operator to a variable that has an undefined scalar subcomponent will be unpredictable, but need not raise an error.

- The value of a scalar variable is undefined after elaboration of the corresponding object declaration unless an initial value is assigned to the variable by an initialization (explicitly or implicitly).

LRM 6.2 Formal Parameter Modes

- The value of a variable is said to be updated when an assignment is performed to the variable, and also (indirectly) when the variable is used as an actual parameter of a subprogram call or entry call statement that updates its value; it is also said to be updated when one of its subcomponents is updated.

- The value of a scalar [out] parameter that is not updated [by a procedure call] is undefined upon return; the same holds for the value of a scalar subcomponent other than a discriminant.

LRM 9.10 Abort Statements

- If the abnormal completion of a task takes place while the task updates a variable, then the value of this variable is undefined.

LRM carefully avoids talk about "defined" or "undefined" or "partially defined" aggregates. The notions "defined" and "undefined" apply only to scalars and scalar subcomponents. No explicit definition is given of what it means for a scalar variable to be defined, other than to say that a scalar variable initialized upon declaration is defined.

The distinction between a declared but null-valued access variable and an unintialized allocation is as follows: if type POINTER is access T, and x is declared to be of type POINTER, then an immediate attempt to evaluate any subcomponent of x, such as x.all, will result in the raising of CONSTRAINT_ERROR (unless the subcomponent appears as prefix to an attribute (LRM 4.1). The value null is in effect an out-of-range index. The result of an attempt to use x in this way is therefore predictable. After executing "x := new T", the result of evaluating x.all is unpredictable, as the value of x is now in-range, but the value of x.all is undefined. In particular an error need not be raised.

The alternative to implementing limited private types (other than task types) as records with default values is to provide an initialization procedure with the package which defines the type. This procedure can be called on immediately after the declaration of any object of that type, but the solution is not fool-proof, precisely because it separates declaration and initialization.

Exceptions raised during declarative parts will cause control to be transferred out of the scope of the variables declared in that declarative part (LRM 11.4.2). No vledgariable declared in that declarative part will linger as an undefined entity because all of them will cease to exist.

The subset proposed by [LEDGARD 82] would eliminate the capacity to provide initializations in declarations, because this complicates implementation, and also eliminate default values. They do not state their reasons for eliminating default values, but presumably it is this: if there are no default values, a programming error which results in failure to initialize a variable is more likely to advertise itself by leading to nonsensical behavior. They seem, therefore, to be addressing a question somewhat different from ours, that of devising a subset which makes <u>testing</u> easier.

We finally note that in our rules for initializing variables, an attempt is made to maneuver around two facts:

- The effect of using an undefined variable is unpredictable, a thing which would be of little concern in itself, since one presumably wants to prevent undefined uses form ever occurring.
- Program text alone cannot easily constrain (legal) compilers to ensure that every variable is defined before it is used.

1.3.2. Floating Point Types: LRM 3.5.7, 3.5.8

The difficulties in verifying floating operations, beginning with the difficulty of stating what one means by correctness, are well known and aren't the sorts of problem to which "restrictions" are the appropriate response. We take as a beginning [SUTHERLAND 84] which formalizes the following notion of the <u>logical</u> correctness of an algorithm: a program is a logically correct representation of a mathematical function if the values which it computes converge to the correct values of the function as the accuracy of the machine on which it is run increases. If a finite polynomial is used to compute some transcendental function such as cos, the correct logical specification of the algorithm would be the assertion that it computes that polynomial, not that it computes cos. At the March 1985 meeting, this proposal, equally applicable (or inapplicable) to any programming language, was criticized on the grounds that the model numbers of Ada are quite carefully specified and might therefore make quantitative analysis of Ada programs tractable. On the other hand, it seems that the straightforward approach (assuming the maximum allowable error in each calculation) rapidly leads to error bounds too large to be useful. [Cohen 85]

1.3.3. Access types: LRM 3.8

The point of this section is simply to rehearse some terminology from LRM and make a few fine distinctions which will be useful later. Consider the following:

```
type T is array(1..2) of INTEGER;
type POINTER is access T;
x,y : POINTER;     — x and y have default access value
                   — null; attempts to evaluate x.all
                   — raise constraint_error
x := new T'(0,0); — x.all is initialized to (0,0);
y := new T'(1,1).
```

The terminology of ARM is: the allocator

"x := new T'(0,0)"

creates an object, and yields, for x, an access value that designates that object. The strings "'(0,0)" and "'(1,1)" initialize x.all, and y.all, the objects designated by x and y. The default initialization of x itself, by the declaration of x, is a special access value null, which does not designate an object. The simple-minded implementation of allocation is that x is assigned an address (the access value), which is the location at which the new object of type T (the object designated by x) resides. The terminology of ARM continues: so long as x contains the same access value it is said to designate the same object, even though the object itself may change. This is reflected in the two kinds of assignment statements. Given the declarations above, the result of:

```
x := y
```

is that x and y designate the same object, because each thereafter will contain the same access value (the one originally contained by y). The result of

```
x.all := y.all
```

is that x and y designate different objects (containing different addresses) whose components happen to be the same: they're identical twins. The result of

```
x(1) := 2
```

is that x designates the same object as before – an object that is now changed (as in, "That's the same man, but now he has a beard").

1.3.4. Slice Assignments and Equality: LRM 4.5.2, 5.2.1

If A and B are array variables, it is possible that the value of "A = B" could be true and the value of "A(2) = B(2)" at the same time false. If the indices of A range from 1 to 5 and those of B from 2 to 6, the "=" operator asks only whether the first value of A equals the first of B, etc. Notice that "A = B" will be true after the slice assignment "A := B".

1.3.5. Annotating Loop Statements: LRM 5.5

We note here a limitation: the specification technique advocated (programmer-supplied "loop invariants") is not generally adequate to specify the behavior of an iterative construct that is not intended to terminate.

1.3.6. Goto: LRM 5.9

As Ada's control structures include "return" and "exit", it seems not unreasonable and not fetishistic to say: no goto's. [LEDGARD 82] argues that goto is "redundant" and [GOOD 80] that it is an "anachronism". The standard general discussion of goto is [KNUTH 77].

1.3.7. Aliasing, Related Names: LRM 6.4, 6.5

"Aliasing" is a general notion, not peculiar to Ada, but the indeterminacy of Ada's parameter-passing mechanisms make the logic of certain aliased procedure calls simply intractable. We wish to rule such calls out. Unfortunately, the general question of whether or not a procedure will be called with two aliassed variables is undecidable. We therefore define a broader notion: whether two variables have "related names" – a question immediately decidable by inspection of the variables.

We begin by giving a precise definition of "alias". Strictly speaking, it applies only to variables from which all identifiers introduced by renaming declarations have been removed. If, however, such variables are seen on the basis of this definition to be aliased, there is no need to return to the original names.

1.3.7.1. Definition of "Alias"

Intuitively, two distinct- occurences of variables are aliases if they refer to overlapping areas of storage. In particular, distinct occurrences of the same variable are, trivially, aliases. (The term "alias" is often restricted to distinct of distinct variables, but it will be convenient here to speak more broadly.)

Arrays and records

We first consider the case of records and arrays. Let A be an identifier which is an array, and suppose the following subcomponent is well-formed: A(t).NEXT(j). That is, A is an array of records, and the objects occupying the record field NEXT are themselves arrays. Combining the terminology of [CARTWRIGHT 81] and [GRIES 80] we'll say that the abstract address of the variable 'A(t).NEXT(j)' in some machine state S is an ordered pair whose first co-ordinate is the identifier "A" and whose second co-ordinate is the selector sequence <value of t, NEXT, value of j> – where the values of t and j are computed in state S, and we may as well think of the field-name 'NEXT' as being its own value, the same in all states.

Let x and y be variables and let their abstract addresses be (I1,s1) and (I2,s2), respectively. Then,

x and y are aliases in state S

if and only if

- I1 and I2 are the same identifier.
- One of the selector sequences s1, s2 is an initial segment (not necessarily proper) of the other (when both are evaluated in state S).

Further,

x and y have related names

if and only if

- I1 and I2 are the same identifier.
- Whenever field names of records occur in corresponding places in the selector sequences s1 and s2, the field names are identical; in other words, we implicitly assume the "worst case" where any pair of corresponding indices denote the same value.

Notice that this definition makes no references to machine states or, in general, to any dynamic information about the program.

Finally,

- A variable is aliased with an expression (has a name related to an expression) only if it is aliased with (has a name related to) a variable which occurs in that expression.

Notice that the selector sequence of the variable A (where A is an identifier) is the empty sequence, which is an initial segment of any other sequence.

It is also very important to notice that "alias" is a semantical notion, while "related name" is purely syntactical.

Examples:

- a(i).NEXT and a(i).NEXT(j) have related names, and are guaranteed to be aliases in all states, and the same is true of a and a(i).
- a(i) and b(i) do not have related names, and cannot be aliases in any state, so long as 'a' and 'b' are distinct identifiers.
- a(j) and a(t) have related names, and they are aliases in those states in which j = t.
- a(i).NEXT and a(i).LEFT do not have related names, and cannot be aliases in any state.
- a(i) and a(i+1) have related names, but cannot be aliases in any state.

The last example shows how crude the notion of "related names" is. It is intended to signal the possibility of aliasing, but in this case issues the warning signal even though aliasing is impossible. That impossibility is based on a semantical fact – that i can never equal i+1. In the general case, comparing a(f(i)) with a(g(i)) or a(g(j)), we won't always know whether aliasing is possible.

Access variables

Access variables are only seemingly more awkward than arrays and records, because the customary notation and terminology disguises their complete analogy with the case of records and arrays. We adopt the terminology and adapt the notation of [LUCKHAM 79]. Suppose that we declare

```
type POINTER is access T;
x: POINTER.
```

We introduce (in imagination) a new object, T*, of a new type (type reference class), with the following intuitive meaning:

- T* is a variable-length array.
- The possible values of indices to T are the values of the variables of type POINTER.
- The possible values of the components of T* are of type T.

T* is a purely theoretical—nothing is added to the language or the program text. The scope of T* is identical with the scope of POINTER. Therefore, in particular:

- T* is a variable which is global to any subprogram declared within the scope of POINTER.

In this new notation the object designated by x, which is denoted in Ada by x.all, is instead denoted by T*(x). If all Ada variables are rewritten in this new notation, then the definition of aliasing used above carries through. Here is another example:

```
type T;

type POINTER is access T;

type T is
        record
                VALUE : integer;
                LEFT  : POINTER;
                RIGHT : POINTER;
        end record;

x,y : POINTER;
```

In the revised notation,

- x.all becomes T*(x).
- x.LEFT becomes T*(x).LEFT.
- x.LEFT.all becomes T*(T*(x).LEFT).
- x.LEFT.RIGHT.LEFT becomes T*(T*(T*(x).LEFT).RIGHT).LEFT.

Notice that in the last example, the selector sequence is not the sequence <LEFT, RIGHT, LEFT> but a sequence of length two:

<T*(T*(x).LEFT).RIGHT, LEFT>.

T* is, essentially, an array of records, and its selectors can have length at most two: the first selector being an access value and the second a field-name. The abstract address of x.LEFT.RIGHT.LEFT is

(T*, <T*(T*(x).LEFT).RIGHT, LEFT>).

Accordingly,

- x and y are not aliased and they have unrelated names.
- x.all and y.LEFT have related names, and will be aliased whenever x = y – for this translates to the assertion that T*(x) and T*(y).LEFT have related names, and are aliased when x = y; this is in full analogy to the case oor arrays and records.
- x.RIGHT and y.LEFT.all have related names, and will be aliased whenever y.LEFT = x – for this translates to the assertion that T*(x).RIGHT and T*(T*(y).LEFT) have related names, and are aliased just in case x = T*(y).LEFT.

One apparent anomaly remains: Suppose that x = y. Although an assignment to x affects neither the value of y nor the value of any of the "subcomponents" of y, an assignment to x.all alters the object designated by y. Yet the definition above says that x.all and y are not aliased – for T*(x) is not aliased with y. The anomaly is psychological: we tend to give x no status of its own, thinking of it as another name for T*(x). We could arrive at the same anomaly in the case of arrays: if A is an array variable and we insisted on thinking of the integer variable i as another name for A(i), we might be puzzled by the fact that an assignment to A would change the value of the object, A(i), "named" by i. What this shows is that calling x.all a "component" of x can in some circumstances be misleading, as misleading as calling A(i) a component of i.

1.3.7.2. Eliminating Parameters With Related Names

It is always possible to "preprocess" procedure calls in a way that guarantees that forbidden pairs of parameters with related names will not occur. It is up to the programmer to find a preprocessing that results in a program that has the desired effect.

The simplest kind of preprocessing is a reassignment: If Q(x..w..) is forbidden because the actual parameters x and w have related names, then one can choose a brand new identifier y, declare it to have the same type as x, and replace the code "Q(x,...)" by "y := x; Q(y,...); x := y." If several variables are so treated, then, of course, the order in which the assignments are made will matter. It is up to the programmer to decide which of these, if any, achieves the desired effect. One of the effects of this trick is to guarantee that the result of the call will be equivalent to the result of a call by copy-in/copy-out.

1.3.7.3. Less Restrictive Rules

Preliminary warning: the "no-aliasing" restriction not only simplifies the input-output semantics of procedure calls, but has the effect of making impossible certain kinds of erroneous assignments to records (which *threaten to change the discriminant in an inconsistent* way). See the discussions of LRM 5.2 and 6.2 in Chapter 4. Therefore, if the "no aliasing" restriction is relaxed, new cautions (not presented here) are needed in handling assignments to records.

Subprogram calls whose actual parameters have improperly related names may be permitted if it can be proven that no execution of the call will occur when those parameters are aliased.

Example 1: It is provable that a(i) and a(i+1) will never be aliased in any program.

Example 2: One can guarantee that a call with improperly aliased actual parameters will not be made by guarding the call with a conditional:

```
if x /= y then Q(a(y),a(x))
else ... something suitable.
```

Notice that guarding a call with a conditional will in general be insufficient if the actual parameter a(x) is potentially aliased with a *global variable* (say, a(z)) *of the procedure*. Whether x equals z at the point of call is irrelevant. What will matter is whether the value of x at the point of call equals the value of z current at certain crucial moments during the execution of the subprogram body.

A subprogram may be verifiably "alias-proof" – that is, it may (verifiably) perform according to expectations even if called on with otherwise forbidden pairs of actual parameters. A standard example is the following procedure, which exchanges the values of two variables.

```
procedure swap(x,y: in out integer) is
    temp: integer;
begin
    temp := x;
    x := y;
    y := temp;
end;
```

It is verifiably the case that calls to "swap" always interchange the values of its actual parameters, whether they are aliased or not, and whatever method of parameter passing is used.

An implementation would be allowed to support pragmas that (either globally or selectively) provide some control over the method of parameter passing, e.g., "pass all aggregates by reference." A procedure may be provably alias-proof for such an implementations even if it is not so for all implementations.

1.3.8. Exceptions: LRM 11

The first difficulty that presents itself, and one whose magnitude can be gauged only through experience, is the possibility that solutions "in principle" will be overwhelmed in practice by a combinatorial explosion in the number of potential paths of control. Every statement is implicitly preceded by conditional jumps associated with each predefined exception; in addition, if more than one exception can be raised, there is no way of telling which one of them actually will be.

Distinctions can be made among the predefined exceptions. Some may be much harder to deal with than others:

- STORAGE_ERROR and NUMERIC_ERROR are implementation dependent, and their occurrence does not necessarily imply a logical error in the program being executed.
- CONSTRAINT_ERROR and PROGRAM_ERROR are "logical" errors, and are independent of implementation; ordinarily one simply wishes to prove that they won't occur – and their logical cleanness seems to make them much more tractable than the first two.
- No one has ever claimed to have any ideas about verifying assertions about TASKING_ERROR.
- IO_EXCEPTIONS (ARM 14.4) and TIME_ERROR (ARM 9.6) are more like constraint_error than they are like numeric_error, but are to some extent machine dependent.

The second major difficulty results from Ada's underdetermined parameter-passing mechanisms. If an exception is raised in a subprogram and not handled there, then the state of its actual <u>in</u> <u>out</u> parameters will depend on whether they were passed by copy (unchanged, because copy-back cannot have occurred) or by referenced (potentially changed). [LUCKHAM 80] present proof rules which allow exception propagation, i.e., permit a subprogram <u>not</u> to handle an exception —- but their rules depend on knowing the mechanism by which each parameter was passed.

Some aspects of Ada's exception mechanisms are discussed in more detail in the next section.

1.3.9. Low-Level Ada: LRM 13

LRM 13.2 says that a representation clause accepted by an implementation must not change the "net effect" of the program. Accordingly, it is true by definition that representation clauses alone present no problem to a verifier. As noted in Chapter 2, when these are used jointly with address clauses to define the interface with ane external device, the main problem seems to consist in specifying the device and the desired interaction with it.

As to the SYSTEM package, we repeat our earlier remark that our proposed restrictions merely reflect the fact that we know of no particular study devoted to it.

1.3.10. Subprograms: LRM 6

The suggested restrictions can be justified not only because they simplify the logic of subprogram calls, but because they help make programs highly modular.

1.3.10.1. Functions

The logic of function calls is simplified if the functions produce no visible external effect other than their output, that is, if they have no side-effects. For example, the standard functions "+" and "*" have no side-effects, and we rely on that. We rely not only the correctness of the value returned by evaluation of "x+y," but also on the assumption that evaluating "x+y" leaves unchanged the values of all the variables in the program. (Note: A serious practical and logical problem corresponds to the big difference between "f(x)" and "x+y": namely, that "f(x)" may not return a value.)

The question arises: what is a side-effect? When a function call is made the program counter moves, the machine's clock ticks, storage fills up, the universe expands. Not everything in the world remains the same. The notion of side-effect is relative. A change is a side-effect only if there is a way in which information about that change is in some way available to the program and can therefore affect its execution. These considerations justify our restrictions on functions.

Changes in the values of variables are visible effects: Therefore, global variables may not be changed, either by assignment by calling on subprograms which change them.

An I/O operation, although not changing the values of any explicit program variables, nonetheless produces a detectable change (e.g., movement of a file pointer), a change detectable by the next call to that operation.

The rule which forbids updating any (or all) of the components of an access variable is a disguised way of forbidding certain kinds of side-effects on global variables. If x is an access variable and a function body contains an update of x.all or to x.LEFT, then (in the notation of the discussion of aliasing) the function body "really" contains an update of T* or T*(x).LEFT – where T* is a variable global to the function.

Notice that the "no side-effects" rule means that the objects designated by access parameters passed to a function will not be changed by the call. On the other hand, objects designated by parameters passed to a procedure, even those passed as in parameters, may be changed by the call.

Allocators are forbidden in function bodies because (as LRM says) an access type implicitly brings into a being a global variable which stands for the totality of allocated objects, and a new statement updates that variable, "incrementing" it by the addition of another object. If x is a local variable of the function, then any object allocated to x is inaccessible after any execution of a call to the function is completed. Nonetheless, such an allocation may leave tracks behind: It may not be cleaned up, and could lead to a STORAGE_ERROR (ARM 11.1).

Notice that merely to read a non-local variable in a function body is to allow external influence on the behavior of that function. Calls on run-time attributes also allow outside influences on the behavior of functions, and in ways that can be much harder to keep track of.

1.3.10.2. Procedures

We provide below a standard illustration of the sort of awkwardness that arises as the result of aiiased procedure calls and note the standard remedy. The discussion of erroneous programs in Chapter 4 will show why the standard remedy is not necessarily helpful in Ada. Consider:

```
procedure P(x: in out INTEGER;
            y: in out INTEGER) is
begin
```

```
            x := y+1;
    end P;
```

We would like to be able to reason about P by enunciating a general principle like this: if x and y are passed to P then, after the call to P, x = y+1. Unfortunately, after the call P(a,a)—a call legal in Ada which, however, violates our "no aliasing" rule—it would seem to "follow" that a = a+1.

The logical mistake is that the proof of the principle "x = y+1" implicitly assumed that the parameters were not aliased against one another, and is not otherwise valid. The standard way to correct the mistake is to verify two separate facts about P, one under the assumption that its actual parameters will not be aliased and another under the assumption that they will. The numbcr of cases goes up rapidly with the number of parameters and the analysis of any one case requires that one know the method of parameter passing the order of copy-out, should parameters be passed out by copy.

Further discussion of aliased procedure calls is contained in the next section (erroneous programs).

1.4. Erroneous Programs, Incorrect Order Dependences, Predictable Compilers

1.4.1. Erroneous Programs

The term "erroneous" is defined in LRM 1.6 as follows:

> The language rules specify certain rules to be obeyed by Ada programs, although there is no requirement on Ada compilers to provide either a compilation-time or a run-time detection of the violation of such rules. The errors of this category are indicated by the use of the word erroneous to qualify the execution of the corresponding constructs. The effect of erroneous execution is unpredictable.

In effect, the compiler is allowed to make certain assumptions about the execution of the program as a basis for generating code, doing optimizations, etc. If those assumptions are violated, the blame falls on the programmer and presumably, the more ingenious a compiler is at exploiting those assumptions, the more peculiar may be the behavior of the compiled code if they are false.

The rules in question occur in Sections 3.2.1, 5.2, 6.2, 9.11, 10.5 11.7, 13.5, 13.10.1, and 13.10.2 of LRM.

LRM 1.6 goes on to say that:

> If a compiler is able to recognize at compilation time that a construct is erroneous or contains an incorrect order dependence, then the compiler is allowed to generate, in place of the code otherwise generated for the construct, code that raises the predefined exception PROGRAM_ERROR. Similarly, compilers are allowed to generate code that checks at run time for erroneous constructs, for incorrect order dependences, or for both. The predefine exception PROGRAM_ERROR is raised if such a check fails.

1.4.1.1. LRM 3.2.1 Object Declarations

An attempt to evaluate a scalar variable which is undefined or to apply a predefined operator to a variable that has an undefined scalar subcomponent is erroneous.

Example 1

```
x, y : integer;
x := y;    -- erroneous, as execution of this
           -- statement requires evaluation of y
x := 0;
```

A compiler could reject this program, or generate code that detects the erroneous step during execution and raises program error at that point, or compile it as is without complaint. The language definition does not permit us to infer that the program will terminate with the value of x equal to zero if it terminates without raising program_error.

Example 2

```
type BA is array (1..10) of Boolean;
x,y : BA
x := y;    -- erroneous?
```

It is not obvious from the text of LRM whether this is erroneous or not. The question comes down to the following: does "evaluation of y" necessarily imply evaluation of its components?

Example 3

The point of this example, taken from LRM 11.6, is that an error can be raised between the declaration of n and its initialization, even though no executable statement appears between them.

```
declare
         n  : integer;
begin
         n := 0;
```

```
        for J in 1 .. 10 loop
                n := n + J**A(k);  -- A and k are
                                    -- global variables
        end loop;
    exception
        when others => PUT(n);
    end
```

LRM says that an implementation may evaluate A(k) before the assignment to n, but not before the begin (as that would associate an error in the evaluation of A(k) with a different handler). If this evaluation raises an exception the handler will attempt to PUT the value of an undefined variable.

1.4.1.2. LRM 5.2 Assignment Statement

An assignment to a variable which is a subcomponent and which depends (as a subcomponent) on the discriminants of an unconstrained record variable is erroneous if any of the discriminants of that unconstrained object is changed by the assignment. (A similar warning is issued in LRM 6.2 about producing such an effect by means of a subprogram call. See the discussion of 6.2, below.) The definition of "depending on a discriminant" can be found in LRM 3.7.1. It is illustrated in the next example.

Example 4a

```
    type ANSWER(LENGTH: INTEGER: = 3) is
        record
            OK: STRING(1..LENGTH); -- the one and
                -- only component, 'OK,' depends
                -- on the discriminant 'LENGTH'
   end record;

    y : ANSWER;   -- y is an unconstrained record variable
         -- and is created with (default) discriminant 3

    c : ANSWER(2) := (OK => "no"); -- c has discriminant 2;

    function f return STRING(1 .. 3) is
    begin
        y := c;   -- allowed: the discriminant of y is
        -- changed by the assignment, but the change is
        -- legitimate because it's done by means of
        -- a complete assignment to all the components
        -- of y
        return "yes";
    end function f;

    y.OK := f;   -- erroneous: y.OK is a subcomponent of y
        -- which depends (as a subcomponent) on the
```

```
             -- discriminants of the unconstrained record
             -- variable y and the assignment changes the
             -- discriminant of y
```

The problem is this: if the language required that f be evaluated before evaluation of the name of "y.OK", then evaluation of the name would return the value "2" as its discriminant and we could be guaranteed that the attempted assignment would not be carried out and would instead raise CONSTRAINT_ERROR. Since the name may be evaluated first, the only way to guarantee that some warning would be raised when y became inconsistent (i.e., its components became inconsistent with its constraints) would be to require a consistency check after the assignment. Rather than impose that run-time burden, the language designers raise a warning in the LRM.

1.4.1.3. LRM 6.2 Formal Parameter Modes

LRM 6.2 (paragraphs 5 and 13) says that an out parameter returns an undefined value if the corresponding formal is not updated by execution of the procedure body. The updating must be done by updating the formal out parameter itself, and not by updating some alias of the actual parameter. A program which updates to evaluate such an undefined variable is erroneous.

Paragraph 10 of LRM 6.2 says:

> If the actual parameter of a subprogram call is a subcomponent that depends on discriminants of an unconstrained record variable, then the execution of the call is erroneous if the value of any of the discriminants of the variable is changed by this execution; this rule does not apply if the mode is in and the type of the subcomponent is a scalar type or an access type.

This is related to the warning in 5.2, but not quite identical. Here is an example:

Example 4b

```
     type ANSWER(LENGTH: INTEGER: =  3) is
             record
                     OK: STRING(1..LENGTH);
   end record;

     y : ANSWER;
     c : ANSWER(2) := (OK => "no");
b : ANSWER(3) := (OK => "yes");
     procedure Q(u: STRING); -- u is an unconstrained
                             -- array variable;
     begin
             y := c;
   u := b;
     end procedure Q;

Q(y.OK); -- erroneous;
```

The point, once again, is that such a procedure call could make y inconsistent. The formal parameter, u, inherits its constraints from the actual, y.OK. It is therefore constrained to be of length 3. Suppose that y.OK is passed by reference. The assignment of c to y changes the object to which u is pointing, making it an array of length 2. If the procedure ended here, u's confusion would cause no problem, but the assignment of b to u does. u "thinks" it is constrained to be of length 3, and that b therefore contains an appropriate value, but if the assignment is allowed, y will become inconsistent. Notice that in this case even a consistency check of y after the procedure call might not be sufficient to avert a mishap: the confused assignment to u might overwrite information in y. We leave it to the reader to see why the manual allows two exceptions.

The "no aliasing" rule will prevent this. Suppose that x.OK is a parameter to a procedure call and some discriminant of x is changed by executing the procedure call. This can occur only by the execution, at some point, of a complete assignment to x, meaning that x itself either occurs in the procedure body or is a global variable to a subprogram called on in the procedure body. That, in turn, means that x must be an actual parameter of the call or a global variable. In either case the call violates the "no-aliasing" rule.

Scalar and access variables must be passed by copy-in/copy-out. The method of parameter passing for parameters of array, record, or task type is up to the compiler (and need not even be the same for successive calls to the same subprogram). The order of copy-in or copy-out is unspecified.

> The execution of a program is erroneous if its effect depends on which mechanism is selected by the implementation. (LRM 6.2)

The word "mechanism" is to be understood broadly, so as to encompass such details as the order in which parameters are copied in or out, etc.

LRM notes a condition sufficient to rule out such erroneous subprograms (under the assumption that the subprogram exits normally), namely, that

- no actual parameter of such a type is accessible by more than one path

i.e., that there is no aliasing. So, LRM disapproves aliasing certain parameters, and we extend that, on logical grounds, to all parameters.

Here are some examples of erroneous programs that result from aliasing. Another example is given in the discussion of shared variables (LRM 9.11).

Let the body of P be:

Example 5

```
procedure P(x: in out INTEGER; y: in out INTEGER) is
begin
     y:= x+1;
end P;
```

The result of the call P(u,u), which violates the rule against having in out parameters with related names, is unpredictable. The initial value of u (call it u0) is copied into both x and y. Executing "y:= x+1" leaves u0 in x and u0+1 in y. The result of copying both x and y back into u will depend on the order in which the copying is done. Notice that we have a problem even though we know (in this case) what the method of parameter passing must be.

Let the body of Q be:

Example 6

```
procedure Q(x: in out ARRAY(1..N) of boolean) is
begin
        x: = not x
        "Search for an i such that x(i) = b(u).  If one
        is found, x: = not x; otherwise, skip."
end if;
end Q;
```

The procedure call Q(b) violates the rules since b(u) is a free variable of Q whose name is related to the actual parameter b. If b is called by copy, then b is changed by the execution of the call. If b is called by reference it's unchanged. Accordingly, the program is erroneous.

Let R be like Q, but with the global parameter made into an explicit in parameter:

Example 7

```
procedure R(c: in boolean;
            x: in out ARRAY(1..N) of boolean) is
```

```
begin
    x: = not x
    "Search for an i such that x(i) = c.  If one is found,
        x: = not x; otherwise, skip."
    end if;
end Q;
```

In the call R(b(u),b) an in parameter is aliased against an in out parameter. Just as in Example 6, the call is erroneous.

1.4.1.4. LRM 9.11 Shared Variables

A shared variable is a variable which occurs in more than one task. A program which violates either of the following restrictions is erroneous (LRM 9.11, paragraphs 4 and 5):

> If between two synchronization points of a task, this task reads a shared variable whose type is a scalar or access type, then the variable must not be updated by any other task at any time between these two points.
>
> If between two synchronization points of a task, this task updates a shared variable whose type is a scalar or access type, then the variable must not be either read or updated by any other task at any time between these two points.

Synchronization is defined as follows (LRM 9.11, paragraph 2):

> Two tasks are synchronized at the start and at the end of their rendezvous. At the start and at the end of its activation, a task is synchronized with the task that causes this activation. A task that has completed its execution is synchronized with any other task.

This series of definitions is poorly worded; taken literally they seem to imply that every point in a task occurs between two synchronization points (the one at the beginning of activation and the one at completion). What is presumably intended is to define something like a matched pair of synchronization points and to require exclusion during the innermost matched pair surrounding a read or update. The reason for the rules is that during a rendezvous, for example, an implementation may keep a local copy of a shared variable and read and write to it rather than reading or writing the shared variable itself.

It is worth pointing out another, perhaps surprising, way in which shared variables can lead to erroneous programs. One might suppose that shared variables would be "safe" if they could be updated only during rendezvous with a special "guardian" task. The example below, boiled down from [WELSH 80], shows that this supposition is false.

Example 8

```
type boolean_array is array(1..1) of boolean;
x: boolean_array := (1 => true);

task resource is
        entry request(u: in out boolean_array);
end;

task type caller;

task body resource is
begin
        loop
          accept request(u: in out boolean_array) do
                u := not u;
          end request;
        end loop;
end resource;

task body caller is
        request(x);
end caller;

caller1, caller2 : caller;
```

Suppose caller1 and caller2 make their calls on resource at roughly the same time, so that caller1 gets accepted and the caller2 is queued. We'll show that the final value of x will depend on the parameter passing mechanism, which means that the program is erroneous. The crucial point is that the execution of an entry call is begun by "any evaluations required for actual parameters in the same manner as for a subprogram call" (LRM 9.5) and the call is suspended to await a corresponding accept. The information will not be recomputed when the accept is made. In our case, the information to be passed from caller2 to resource is gathered when caller2 makes its call, and before caller2 is suspended. If the parameter passing mechanism is copy in/copy out that information includes the then-current value of x, and if the mechanism is pass-by-reference, it includes the address of x.

Suppose first that the parameters are passed by copy-in/copy-out. Then the value passed to resource for caller2's rendezvous will be the stored value (true) and the fact that the present value of x will be (false) (having been changed during the rendezvous of resource with caller1) is irrelevant. When tasks caller1 and caller2 terminate, the value of x will be false.

If, on the other hand, the parameters are passed by reference, then resource will receive the address of x on rendezvous with caller2 and when that rendezvous occurs, the address will contain the value false. Accordingly, the value of x will be (true) when the calling tasks terminate.

1.4.1.5. LRM 11.7 Suppressing Checks

If checks on constraints, overflows, etc., are suppressed and the constraints, etc., violated by an execution of the program, then that execution is erroneous. As indicated in the discussion of exceptions, a verification is likely to accumulate in passing enough information to show that constraint checks and checks for program errors can be safely suppressed.

1.4.1.6. LRM 13.5 Address Clauses

An address clause resulting in overlaying an object or program unit or linking an interrupt to more than one entry is erroneous. Whether an address clause results in overlaying an object is entirely implementation dependent, and verifications of programs with address clauses are non-portable. One might verify such programs under the assumption that this error did not occur.

1.4.1.7. LRM 13.10.1 Unchecked Programming

Use of unchecked deallocation can lead to dangling pointers, and an attempt to access the objects which such pointers designate is erroneous. Once again, a verification would presumably provide, in passing, a proof that no such attempts occur. In the absence of such a proof, unchecked allocation may not be used.

1.4.2. Incorrect Order Dependences

LRM states that certain steps in execution (or elaboration, or evaluation) occur "in some order that is not defined by the language" and that programs which depend on the order of execution (or elaboration, or evaluation) are incorrect. A program can contain incorrect order dependences, as a result of side-effects, either in functions or in the bodies of packages. This is therefore another reason to restrict constructs which cause side-effects.

The sections of LRM that discuss and define the incorrect order dependences are 1.6, 3.2.1, 3.5, 3.6, 4.1.2, 4.3.1, 4.3.2, 4.5, 5.2, 6.4, 10.5.

Consider the following sequence of declarations:

Example 9

```
package A is
     I: integer := 1;
end A;
```

```
package body A is
     I := 0;
end A;

with A;
package B is
     J: integer := A.I;
end B;
```

The rules for elaboration require that the specification of A be elaborated before both the body of A and the specification of B, but require nothing further of the order of elaboration. Should the specification of B be elaborated before the body of A the value of BJ will be 1, and otherwise it will be 0. This is an incorrect order dependence.

All further points about incorrect order dependences can be made fully by looking at one further example, (LRM 3.5): when elaborating a range constraint, the simple expressions specifying the bounds are evaluated "in some order not specified by the language." Let the range in question be f(m)..g(n). Here are two cases in which it will matter whether f(m) is evaluated first or second:

- If a call to the function g alters the value of m.
- If the result of a call to f can be affected by the fact of a previous call on g.

In each case the call on g has a side-effect. The kind of side effect seen in the first case has already been ruled out by the restrictions placed on the definitions of functions. The rules on functions also eliminate the obvious way to generate an example of the second case by letting the value returned by f depend on some global variable i and letting any call of g increment i by 1. But the obvious way is not the only way. It's possible to record the fact that g has been called without storing anything in a variable. Here is an example:

Let T be a task with the single entry ENTER, whose sole action consists of the following: Accept ENTER and then terminate. Let F and G be functions with the same body, namely:

Example 10

```
        x: INTEGER
        begin
                if T'TERMINATED then x := 1
                else ENTER;
                         x := 2;
                end if;
                return x;
        end;
```

The value of F(1) - G(1) will be +1 or -1 according to whether F(1) or G(1) is evaluated first.

1.4.3. The Classification of Program Errors

The classification of errors into erroneous programs, incorrect order dependences, and program errors which are "none of the above" is a practical, not theoretical, classification. Consider these three examples:

- Erroneous: The result of a procedure call depends on the order in which values are copied back into actual parameters (Error 1).
- Incorrect order dependence: The result of evaluating an expression depends on the order in which its terms are evaluated (Error 2).
- None of the Above: An attempt is made to instantiate a generic unit before its body has been elaborated a program error which is neither erroneous nor an incorrect order dependence (Error 3).

Each of these unpleasant occurrences is in some sense an error about the order of events. They are classified on the basis of what it is reasonable to expect of compilers and run-time support. Error 3 must be detected at run-time and the exception PROGRAM_ERROR must be raised. Errors 2 and 3 need not be detected at run-time, and if they are detected PROGRAM_ERROR need not be be raised (but ,ay be). The distinction between 1 and 2 is that: the effect of error 1 is completely unpredictable, while the result of the expression evaluation described in 2 will be either the raising of PROGRAM_ERROR or the computation associated with some particular choice of the order of execution (LRM 1.6).

1.4.4. "Clusters" and "Predictable" Compilers

1.4.4.1. Examples

We begin by collecting a few observations made in previous sections and adding one new example. The numbered examples referred to are the ten numbered examples, Section 1.4.1.1. In order to avoid confusion we will therefore denominate our examples with letters, rather than numbers.

Observation A. We would be able to predicte the effect of code generated from example 1 if we knew that the compiler generated code satisfying any one of the following three assumptions:

* The exception PROGRAM_ERROR is raised at run-time whenever an undefined variable is read.

* The exception PROGRAM_ERROR is never raised at run-time when an undefined variable is read. Furthermore, code generation does not depend on the assumption that all variables will be defined before being used.

* The normal peep-hole optimization has been made, of "erasing" the useless statement "x := y."

If a compiler satisfies one of the first two hypotheses we'll say that it "consistently raises read errors."

Observation B. The effect of Example 3 could be made predictable if a pragma were available by which one could mark off a stretch of code and insist that it be executed in precisely the order given, and with no other steps of execution interpolated. We could thereby insist that nothing extra took place between the declaration of the variable n and its initialization. Such a pragma, coupled with systematic rules for its use, could be substituted for the elaborate restrictions outlined in Section 2, and could solve problems (such as the problem of ensuring that out parameters always receive values) which the method of Chapter 2 cannot. We will say that such a pragma calls on the compiler "to respect the text."

Observation C. The results of the erroneous procedure calls in Examples 6, 7, and 8, but not that of Example 5, would become predictable if the calls were prefaced by a pragma stipulating that all aggregates were to be passed by reference. The result of Example 5 could be made predictable by a pragma that, for example, stipulated a left-to-right order of copy-out (if there are no side-effects, the order of copy-in is irrelevant).

Note 1: Pragmas allowing the programmer to choose the method for passing each parameter are not forbidden by the LRM, which requires only that pragmas not affect the legality of programs (LRM 2.8). An illegal program is one which a compiler must reject, and a compiler need not reject erroneous programs. Such pragmas, however, seem to be a serious break with the principles of Ada, while the one described previously is more modest.

Note 2: As indicated above, allowing aliasing removes an automatic protection against other kinds of erroneous programs. See the discussion of LRM 5.2 and LRM 6.2, in the first half of this section.

Observation D. Consider the program fragment:

```
declare
        type T is range 0..20;
        subtype small_T is T range 0..10;
        x : T;
        y : small_T;
begin
        x := 10;
        <statement>;
end
```

Consider three possible instantiations of <statement>:

```
x := (x+15) - 10; -- NUMERIC_ERROR may be raised;
                     CONSTRAINT_ERROR may not;

x := x + 15; -- NUMERIC_ERROR must be raised;

y := 2 * x; -- CONSTRAINT_ERROR must be raised;
```

These requirements come from ARM 3.5.4 and 11.6.

LRM 3.5.4 says that the possible values of type T include at least the values 0 through 20. An implementation must choose one of its predefined integer types having a range at least as great as 0..20 and implement T as a subtype of that predefined type. The intermediate value "x+15" lies outside the range 0..20 – but if that value is within the range of the predefined type acting as the anonymous base type of T there will be no hardware event (no overflow) to signal an anomaly, and the language definition does not require checks of such intermediate values to see whether they lie in the range of T. On the other hand, if "x+15" were outside the range of the anonymous base type, hardware might signal an error. The upshot is that NUMERIC_ERROR is allowed, but not required. A range check must occur, however, when the computation is completed and the assignment attempted. If the final value is outside the explicitly declared type the error to be raised is NUMERIC_ERROR, and if outside an (explicitly declared) subtype, the error to be raised is CONSTRAINT_ERROR.

To make this more systematic, one might require the following of all predictable compilers:

- The anonymous base type of all integer types will be the type INTEGER.
- NUMERIC_ERROR will be raised by intermediate computations if and only if they overflow the bounds of type INTEGER.

We'll call this "consistent implementations of integer types."

Note: This restriction is far from a sufficient solution to the logical difficulties surrounding the raising of NUMERIC_ERROR.

1.4.4.2. Some Crude Clusters

To suggest how one might carve out tractible Ada clusters, and how they might be related to compiler (or, more generally, implementation) restrictions, we informally describe a few such combinations:

- We will insist that all compilers provide consistent implementations of integer types.

Cluster a1:

- Language restrictions

- Procedure calls must be provably non-aliased (in particular, it would suffice to satisfy the "non-aliasing" rules set out in Chapter 2).

- All exceptions must be handled locally. This requires, in particular, a demonstration that no unhandled exception occur in an exception handler.

- Implementation restrictions

- Impose no restrictions on a compiler's freedom to choose parameter mechanisms.

Cluster a2:

- Language restrictions

- The form of procedure calls will be unrestricted.

- Rules will require certain uses of the "parameter" pragma (see implementation restrictions).

- Exceptions may be propagated from procedures (provided the proper use is made of the "parameter" pragma).

- A new set of restrictions would be required on assignments to variant records.

* Implementation restrictions

- The implementation must support pragma which forces all aggregates to be passed by reference.

Cluster b1:

* Language restrictions

- The severe restrictions of Chapter 2 will be imposed on declartions and initializion of variables.

* Implementation restrictions

- Compilers are allowed a free hand at re-ordering computations.

- The implementation must must raise read errors consistently.

Cluster b2:

* Language restrictions

- Declarations and initializations are not restricted, except for the need to handle the problems raised in example 9, in section 1.4.2.

- Rules for the use of the "respect_text" pragma must be followed.

* Implementation restrictions

The implementation must support a pragma requiring that the text be respected, i.e., restricting the possibility of re-orderings.

- The implementation must raise read errors consistently.

One could also make a cluster from any union of one from column a with one from column b.

1.5. Bibliography

[ANDREWS 83] Andrews, G.R. and Schneider, F.B., "Concepts and Notations for Concurrent Programming," ACM Computing Surveys 15,1 (Marh 1983): 3-43.

A useful survey, from the earliest proposals for concurrency constructs (such as fork and join) to the most recent. These constructs are organized into three general classes: procedure-oriented, in which shared variables are used for communication and the principal problems are mutual exclusion and synchronization of conditions (by semaphores, for example); message-oriented, in which communication consists of the sending and receiving of messages and the principal problems are synchronizing communications and designating the sender and receiver; and operation-oriented (the "remote procedure call" such as Ada's rendezvous), seen as a coming together of the other two. An extensive bibliography is included.

[APT 81] Apt, K.R. "Ten years of Hoare's Logic: A Survey – Part 1," ACM TOPLAS 3,4 (October 1981): 431-483.

A systematic account relating Hoare-style axiomatics to a precise formal semantics for languages with: while, procedures (recursive procedures are included, and a variety of parameter mechanism), arrays, and 'procedures as parameters.' It is self-contained, there are no known errors, and the bibliography is extensive.

[APT 80] Apt, K.R., Francez, N., and deRoever W.P., "A Proof System for Communicating Sequential Processes," TOPLAS 2,3 (July 1980): 359-385.

This paper contains a proof system for CSP (see [Hoare, 1979]). It is a Hoare-style logic for partial correctness which can also be used to prove programs deadlock-free. Proofs of soundness and completeness are mentioned but not provided. The separate concurrent processes can first be reasoned about individually: they are specified as Hoare-triples and proved on the basis of explicit assumptions about the inputs supplied by other processes at the rendezvous's. The processes must then be shown to co-operate – to be consistent with the assumptions made about them in the first stage.

[BARRINGER 82] Barringer, H. and Mearns, I., "Axioms and Proof Rules for Ada Tasks," IEE Proceedings 29/Part E, 2 (March 1982): 38-48.

An adaptation of [Apt, Francez, and de Roever 1980] to some of the tasking features of Ada. Shared variables are not allowed, nor are subprograms with side effects. Proposals are also made for extending this beyond the CSP-like features of Ada tasking: to nested accepts, delay statements, conditional entry calls, timed entry calls, selective waits with else-parts, and tasking errors. Neither formal semantics nor a soudness proof are presented or claimed.

[CARTWRIGHT 81] Cartwright, R. and Oppen, D., "The Logic of Aliasing," Acta Informatica 15 (1981): 365-384.

A formal semantics and proof system are proposed for procedure calls in a modified Pascal-like language allowing array types and aliasing, and obeying the following restrictions: 1. No functions may be passed as parameters. 2. Every global variable accessed in a procedure must be accessible at the point of every call. 3. No procedure named p may be declared within the scope of a procedure p. The paper is very difficult to read. Someone wishing to attempt it should first read [Gries and Levin, 1980].

[COHEN 85] Cohen, N., "Axiomatic Semantics for Ada," a talk given at the Ada Verfication Workshop, March 18-20, 1985, at IDA.

[COOK 78] Cook, S.A. "Soundness and Completeness of an Axiom System for Program Verification," SIAM J. Computing 7,1 (1978): 70-90.

This paper sets out the now-standard theoretical definition of the meaning of "completeness" for Hoare-like axiom systems, namely, that a system is complete if it is complete relative to the complete theory of the underlying domain of data types (and assuming also that the language of the underlying domain meets a certain technical condition called "suffficient expressiveness").

[DEBAKKER 80] deBakker, J.W., Mathematical Theory of Program Correctness, Prentice-Hall, 1980.

A microscopic account, all details provided, of the denotational semantics of a variety of combinations of sequential programming constructs (including: while, recursive procedure calls, blocks, go to).

[FLOYD 67] Floyd, R., "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science, American Mathematical Society, (1967): 19-32.

The original presentation of the "Floyd" half of "Floyd_Hoare."

[GEHANI 84] Gehani, N., Ada: Concurrent Programming, Prentice-Hall, 1984.

A lucid exposition of Ada tasking, with many examples.

[GERTH 82] Gerth, R. "A Sound and Complete Hoare Axiomatization of the Ada Rendezvous," Proc. 9th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 140, Springer Verlag (1982): 252-264.

An adaptation of [Apt, Francez, and de Roever 1980] to a partial correctness logic for a fragment of Ada tasking. The fragment in question is defined precisely: the types are boolean and integer; the sequential statements are while, if, and assignment; the set of tasks is fixed and all are activated simultaneously; tasks may not have shared variables; the allowed communications statements are calls (not conditional calls), accepts, and selective waits (without else-parts or delays); there are strong "no-aliasing" restrictions on the parameters of an entry call (thereby justifying an assumption of call-by-copy semantics). Some details of Ada semantics are modified: there are no entry queues (a partial correctness logic is unable to distinguish between this and fairness); calling on a terminated task leads to deadlock, not an error. Soundness and completeness proofs are sketched: they proceed by translating programs from the Ada fragment into CSP programs and using the completeness result of [Apt, Francez, and de Roever, 1980].

[GERTH 83] Gerth, R. and deRoever, W.P., "A Proof System for Concurrent Ada Programs," RUU-CS-83-2, Rijksuniversiteit Utrecht, January 1983.

An extension of [Gerth, 1982] to deal with proofs of safety properties deadlock freedom, and termination. Calling on a terminated task is now treated, as in Ada, as an error. The authors observe that they can trivially extend their system to incorporate delays, conditional and timed entries, and conditional and timed accepts for the trivial reason that the effects of such calls aren't expressible in the assertion language. The authors remark that their approach depends essentially on the assumption of a fixed number of tasks, activated simultaneously, and on forbidding queue attributes and access variables to task types.

[GOOD 84] Good, D.I., "Revised Report on Gypsy 2.1 (Draft), July, 1984."

[GOOD 80a] Good, D.I. and Young, W.D., "Generics and Verification in Ada," Proceedings of the ACM Sigplan Symposium on the Ada Programming Language (1980): 123-127.

The principal observation of this paper is that one can't expect to verify the behavior of generics with respect to completely arbitrary instantiations and, accordingly, some mechanism must be provided for restricting the parameters of the instantiation and specifying those restrictions. One of the shortcomings the authors note has subsequently been attended to. In the final version of Ada it is possible, for example, to restrict the instantiation of a generic type solely to integer types, or solely to discrete types, etc.

[GOOD 84b] Good, D.I., Young, W.D., and Tripathi, A.R., "A Preliminary Evaluation of Verifiability in Ada," Proceedings of the 1980 Annual Conference of the ACM (1980): 218-224.

What the title suggests. The authors are commenting on preliminary Ada. The final version of Ada is, from the point of view of their criticisms, an advance on some fronts and a retreat on others.

[GRIES 80] Gries, David and Levin, Gary, "Assignment and Procedure Call Proof Rules," ACM TOPLAS 2,4 (1980): 564-579.

Proof rules are proposed for procedure calls in languages containing array and record types and in which: formal parameters are specified as var, result, or val; global variables are allowed in procedure bodies. Aliasing is not allowed, but specific instances of aliased calls can be accommodated by rewriting the given aliased call as an unaliased call to a related procedure. No formal semantics is provided (and therefore no soundness proof), but the intuition behind the rule is clearly presented.

[GRIES 79] Gries, David and Owicki, Susan, "Verifying Properties of Parallel Programs: An Axiomatic Approach," Communications of the ACM 19,5 (May 1979): 279-285.

An earlier version appeared as TR 75-243, from Cornell University. An early axiomatic method for proving properties of parallel programs is presented. The parallel construct considered is cobegin-coend. A second, delimiting a "critical section," is used for synchronization and protection of shared variables with r when B do S.

[HOARE 69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM* 21,8 (1969): 578-580, 583.

The original presentation of the "Hoare" half of "Floyd-Hoare."

[HOARE 71] Hoare, C.A.R., "Procedures and Parameters: An Axiomatic Approach," in *Symposium on Semantics of Algorithmic Languages*, edited by Engeler, Lecture Notes in Mathematics, Volume 188, Springer-Verlag (Berlin 1971): 102-116.

This paper extends [Hoare, 1969] by adding a rule for procedure calls (including recursion). The problems with aliasing are illustrated and it is observed that if the proof rule is taken as the definition of the semantics of procedure calls, then programs using only unaliased calls can be correctly implemented by any of the standard mechanisms for. parameter passing. No formal semantics is provided.

[HOARE 78] Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM* 21,8 (1978): 666-677.

This paper proposes a construct which is the ancestor of the Ada rendevous. CSP is a toy language having as sequential constructs assignment, iteration, guarded alternatives. A cobegin statement may activate a fixed set of (non-nested) parallel processes simultaneously, and control may not pass beyond the cobegin until all processes have terminated. The processes may communicate only through paired input-output statements that have the effect of an Ada rendezvous in which parameters are passed but the accept body is empty.

[ICHBIAH 79] Ichbiah, J. *et al*, "Rationale for the Design of the Ada Programming Language," *SIGPLAN Notices* 14,6 (June 1979): Part A.

Note that many of the features of Ada discussed in this report have since been changed.

[KNUTH 77] Knuth, D.E., "Structured Programming with Goto Statements," in *Current Trends in Programming Methodology*, *vol. 1*, R. T. Yeh, ed., Prentice-Hall (1977): 140-194.

From the paper's introduction: "This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs efficiently without go to statements; and (b) a methodology of program design, beginning with readable and correct, but possibly inefficient, programs that are systematically transferred if

necessary into efficient and correct but possibly less readable code."

[LEDGARD 82] Ledgard, H.F. and Singer, A., "Scaling down Ada," Communications of the ACM 25,2 (February 1982): 121-125.

Suggestions for a standard Ada subset. The authors have several goals: to make the language easier to learn, implement, and standardize; to make standardization and validation easier; to pre-empt ad hoc subsetting; to reduce the likelihood of programmer errors.

[LUCKHAM 80] Luckham, D.C. and Polak, W., "Ada Exception Handling: An Axiomatic Approach," ACM TOPLAS 2,2 (April 1980): 225-233.

Proof rules are presented for Ada exception-handling which are adaptations of the standard rules for goto. The authors note that applying their method to the predefined exceptions requires something like the insertion of several implicit goto's after every program step – one for each exception which could be raised by its execution – and that this may well increase the computational costs to impractical heights. (The problem is more difficult than handling goto's because the target of the jump may not be statically determinable.) To deal with exceptions raised during the execution of procedures it is in general necessary to know the methods used for parameter passing. Tasking exceptions are not covered.

[LUCKHAM 79] Luckham, D.C. and Suzuki, N., "Verification of Array, Record, and Pointer Operations in Pascal," ACM TOPLAS 1,2 (October 1979): 226-244.

Proof rules are provided for the operations of assignment, selection, dereferencing, and allocation. Extensions are proposed to the standard rules for a fragment of Pascal which would incorporate procedure calls with pointer variables as actual parameters, etc. Pointer operations are modelled on arrays, which are already well-understood. Pointer variables are though of as indices to an "array" and dereferenced pointers as the components of the "array." Allocation adds to the range of the "array"'s allowed indices. The authors refer to, but do not provide, proofs of the soundness and completeness of their rules, but it is not made clear with respect to what assertion languages. A correction to their allocation rule is noted in [Gries and Levin, 1980]. The authors note that reasoning about complex (especially: recursive) data structures requires additional notions, such as "reachability". A list of 20 axioms is provided for the notions of "reachability" and "betweenness". Sample proofs – both proofs by hand and automated proofs – are provided.

[LUCKHAM] Luckham, D.C., von Henke, F.W., Krieg-Bruecken, B., and Owe, O., "ANNA, a Language for Annotating Ada Programs, Preliminary Reference Manual," Stanford Computer Systems Lab technical report 84-261.

A report on the most substantial effort known to us for producing a specification language for Ada. Annotations are by and large generalizations of the Ada notion of "constraint". An annotation may, for example, constrain all variables of some integer type to have even values (a constraint which can't be made in Ada). The annotator can control the scopes in which annotations are to hold – and, in particular, what is usually called an "embedded assertion" is an annotation whose scope is a single point in the text. An annotated program (possibly containing specially marked off auxiliary code, called "virtual text") is to be translated into an "ANNA kernel", a new Ada program in which, for the most part, the annotations have been rewritten as embedded assertions. If, for example, an integer type is annotated as having only even values, every place in the program at which a variable of the given numeric type could be altered would be followed by an embedded assertion saying "the value of the variable is even." Such a kernel could be executed – with the truth of the embedded assertions being tested wherever they arose – or run through a verifier which would generate verification conditions, etc. There is no formal semantics for ANNA. Instead, its semantics is defined by the reduction of ANNA programs to the ANNA kernel – in effect, to Ada semantics.

[MANNA 81] Manna, Z. and Pneuli, A., "Temporal Verification of Concurrent Programs: The Temporal Framework for Concurrent Programs," in The Correctness Problem in Computer Science, ed. R.S. Boyer and J.S. Moore, Academic Press, 1981

This paper sets out a system of temporal logic, a modal logic suitable for expressing and reasoning about certain properties of ordinal (non-quantitative) discrete time. A formal semantics for temporal logic is provided and a variety of assertions are shown to be semantically valid. Execution of concurrent sequential processes is modelled by interleaving the steps in their execution, and it is then shown that many interesting properties of concurrent computations are expressible in the notation of temporal logic: invariance properties (stating that some condition always holds true), eventuality properties (stating that if condition A occurs then condition B must eventually become true), and precedence properties (stating that one event must precede another.

[MCGETTRICK 82] McGettrick, Andrew J., Program Verification Using Ada, Cambridge

University Press, .1982.

An informal textbook aimed at advanced undergraduates. The illustrative language is not the usual pseudo-Pascal, but Ada.

[ODONNELL 82] O'Donnell, M.J. "A Critique of the Foundations of Hoare-Style Programming Logic," Communications of the ACM 25,12 (December 1982): 927-935.

This paper shows that the failure to demand a correct definition of "correctness" has filled the literature with "proof systems" which are inconsistent outright, or are unsound in the sense that the addition of true axioms can make them inconsistent. On the way to true conclusions these systems in effect indulge in a kind of trick -- intermediate inferences which are illegitimate, and lead to trouble as soon as there are enough truths available in the system to exploit their weaknesses. The correct definition of "correctness" is not "every theorem is true" but "everything inferred from a truth is true."

[OLDEROG 81] Olderog, E.R., "Sound and Complete Hoare-like Calculi Based on Copy Rules," Acta Informatica 16 (1981): 161-197.

A systematic treatment of procedure calls is given for a variety of Algol-like languages, with various scope rules, which allow procedures as parameters. The author is primarily interested in characterizing the languages for which his calculi will be complete. Although this is in some sense the fullest and most treatment of procedure calls, it does not help solve the problems encountered in treating procedure calls in Ada.

[OLDEROG 84] Olderog, E.R., "Hoare's Logic for Programs with Procedures -- What Has Been Achieved?" in Logics of Programs, 1983, Lecture Notes in Computer Science no. 164, ed. E. Clarke and D. Kozen, Springer-Verlag, 1984.

A survey which is emphatically not an introduction.

[PNEULI 82 Pneuli, A. and deRoever, W.P., "Rendezvous with Ada -- a Proof Theoretical View," Proc. AdaTEC Conference on Ada Arlington, Va. (October 1982): 129-137.

An operational semantics is defined for an informally described fragment of Ada, using interleaved execution to model concurrent execution. It is then shown that for any program written in this fragment and not using the queue attribute COUNT partial correctness semantics cannot distinguish between: (a) putting entry calls into a fifo queue, and (b) selecting non-deterministically but "fairly" from the waiting calls. A system of temporal logic is defined for making assertions about programs over this semantics and various proof rules are shown sound. A program in the fragment is a block containing a fixed number of tasks. Within tasks: there may occur no subprograms or nexted blocks; there may be no delay statements; selective-wait alternatives may only be accept-alternatives or terminate.

Stanford Verification Group, "Stanford Pascal Verifier User Manual," STAN-CS-79-731, March 1979.

This report describes the use of the PASCAL verifier. Practically all of PASCAL is handled. "Only some of the theory [of data structures] is implemented by the simplifier and it is up to the user to include in his rulefile rules ... to express any required data structure axioms."

[SUTHERLAND 84] Sutherland, D., "Formal Verification of Mathematical Software," NASA contract report 172407, Odyssey Research Associates, 1984.

This paper presents a definition of logical correctness for floating point computation – the "asymptotic paradigm". It says, intuitively, that a numerical algorithm is logically correct if its outputs can be made to converge more and more closely to the mathematically correct value by running it on more and more accurate machines. The semantics is formalized using non-standard models of the real line.

[WEGNER 83] Wegner, P. and Smolka, S.A., "Processes, Tasks, and Monitors: a Comparitive Study of Concurrent Programming Primitives," IEEE Transactions on Software Engineering SE-9,4 (July 1983): 446-462.

As the title indicates, CSP, Ada, and monitors are compared at work on several standard concurrent applications.

[WELSH 80] Welsh, J. and Lister, A., "A Comparative Study of Task Communication in Ada," Software Practice and Experience 11 (1980): 257-290.

Ada is compared to CSP and to Distributed Processes.

2. ADA VERIFICATION IN THE NEAR TERM

This chapter considers what progress in Ada verification can reasonably be hoped for in "the near-term", i.e., within two years. The first section discusses three proposals for near-term verification system in some detail and the second section discusses standards. The question of specification languages is reserved for a separate chapter.

2.1. Near-Term Verification

We know of no existing verification system for Ada programs, and such a system cannot be built from scratch in the near term. We will review current proposals and current work toward adapting existing tools to Ada, and also briefly discuss some experimental systems of which we have some knowledge and which might be thus adaptable.

2.1.1. An Overview

2.1.1.1. Cornell Synthesizer-Generator

Odyssey Research Associates has proposed the use of the Cornell Synthesizer Generator (CSG) as the basis of a near-term Ada verification environment. The CSG can be thought of as a system which generates tools directly from their specifications – provided those specifications are be expressed in the form of an attribute grammar. For example, verification conditions generated by Floyd-Hoare style axioms can be specified in this way. The CSG can therefore accept as input an axiomatic semantics (properly expressed) and will return a VC generator. One is free to choose the specification language. Obvious choices are ANNA, or some language for which one has available an existing theorem-prover or proof-checker. The intention of the project is to study piecemeal verification, dividing the Ada language into many overlapping, but individually tractible subsets, and requiring that any program unit be written solely within the confines of one subset. A paper on this proposal is included.

2.1.1.2. Gypsy

The three best known and most extensively used systems for verifying design or code are Ina Jo, HDM, and Gypsy. We know of no attempts to adapt Ina Jo or HDM to Ada.

J. McHugh (Research Triangle Institute) and K. Nyberg (Verdix Corporation) have considered adapting the Gypsy Verification Environment. They propose to do the following: model an appropriate subset of Ada in the Gypsy language; use the front end of the validated Verdix compiler as front end to their system; and use a modification of the Gypsy Verification

Environment as its back end. A paper of McHugh and Nyberg is included in Appendix A.

2.1.1.3. Modula

The Case Verifier has been developed at Case Western Reserve University for verifying programs written in Modula. Modula comes equipped with the notions of "module" and "process" that are analogous to Ada's "package" and "task". Ernst, Hookway, and others at Case Western have done research into adapting this to Ada, and a paper by Hookway describing this work is included in Appendix A.

2.1.1.4. AVID

AVID, like the CSG, can function as a tool generator. In the course of implementing the AVID verification system a high-level description language, called TDL (template description language), was developed. The syntax for Ada can easily be put into the AVID system to generate a whole new system for writing provably correct Ada programs.

TDL's high-level descriptions make it possible to avoid understanding the low-level implementation (in the case of AVID, the programming language C). The code of AVID itself need not be modified.

Like the CSG, AVID would also produce as a byproduct a syntax directed editor for Ada, making syntax errors impossible, and clearly indicating the causes of semantic and verification errors. In addition, AVID has a fairly powerful deductive apparatus (the assertion table from PL/CV). Although not as powerful as a theorem prover, it is more directly under the control of the user, and therefore requires mathematically sophisticated users.

There are two large disavantages to using AVID. First, TDL is a more complicated interface than first-order logic (as used in the CSG). Second, AVID is unsupported and experimental software.

2.1.1.5. PRL

AVID was a precursor of PRL. The PRL (program refinement logic) project has developed a flexible tool for proving theorems, and bills itself as an attempt to "implement mathematics". We consider here the possibility of using PRL as a "mathematician's apprentice". To use PRL in this way one must first represent Ada programs by suitable mathematical objects, although it would not be necessary to represent the whole Ada language.

Once this has been done (a non-trivial task) any theorem prover is a candidate to be the logical workhorse. The merits of PRL are as follows: PRL has a rich type structure, is well-developed and flexible, allows the user to program proof strategies ("tacticals") for repeatedly performing similar chains of reasoning, has facilities for managing libraries of theorems and for introducing new notation, and has lots of windows and menus (and mouse commands). There is reason to hope that the front-end translator, which would translate Ada programs to their representations (as formalized in PRL) could be fairly simple.

The major advantage and major disadvantage of PRL are one and the same: most of the work would lie in designing the representations for Ada programs. This work is all mathematical, which is good, since no work has to go into writing or modifying software. But the possibilities for at least moderate success are less certain. The success or failure of this approach would depend on how natural the representations proved to be.

Using PRL as suggested raises a problem inherent in all "denotational" approaches. The denotation of a program contains all the information about the program, most of it irrelevant to the goal at hand. A proof about a while-loop really needs to know only a suitable loop invariant.

2.1.2. Three Proposals

2.1.2.1. A Near-Term Ada Test Bed

No one expects that verification of the whole Ada language will ever be possible. Instead of attempting to identify a single verifiable subset of Ada, one should attempt to identify many (overlapping) verifiable subsets. From the programming language CLU [LISKOV 77] we borrow the term "cluster", to denote a tractible (verifiable) subset of the Ada language.

Ada was not designed to be inherently verifiable, and many of its constructs interact in ways that surprise even its designers. By using Ada's mechanisms for modularization and information hiding, it may be possible to write and verify programs using large parts of the Ada language, so long as each program unit is written in a single cluster. Co-ordinating with the notion of a tractible cluster is that of a "predictable" compiler, a valid Ada compiler whose behavior is more deterministic than strictly required by the language definition. For example, the interactions among procedure calls, exception handling, parameter mechanisms, and optimization (the last two depending on choices of the compiler) are very complex, and a variety of clusters could be made from a variety of trade-offs among them.

This strategy is a crude step toward exploiting the fact that in actual programs constructs are not thrown together haphazardly, but occur in contexts. Clusters are meant to be abstract representations of such contexts.

Modularizing the proof system in this way has certain practical advantages:

- Questions of technique will not be prejudged. Nothing requires that different clusters be attacked by the same methods, or by methods that could easily be integrated with one another if they had to be used in tandem on the same cluster.

- The system can be improved piecemeal. Note: There is no reason to think that the all improvements to such a system would consist of enlargin clusters. Subsetting a cluster could be desirable, if the resulting subset were itself useful and could be handled much more easily than its parent.

The discovery of appropriate clusters is largely an empirical affair. To support such experimentation we feel that a near-term solution which provides an Ada verification test bed is essential. We describe our approach to this problem, via the CSG.

The CSG, described in [TEITLBAUM 81], was originally a teaching aid which allowed a student to construct a PL/I program on the basis of user commands corresponding to abstract syntax. The Synthesizer filled in all the concrete syntactical details so that it was impossible to write a non-parseable program. The Synthesizer was subsequently generalized so that it would be applicable to a variety of programming languages. The current CSG [TEITLBAUM 84] is reconfigurable around any user supplied attribute grammar and has been harnessed for a wide variety of purposes.

A recent paper by one of the developers of CSG describes how it can be adapted to generating verification conditions [REPS 84]. Our plan is to use CSG to generate a near term program verification environment for Ada. The essential idea is to augment an attribute grammar for Ada by additional semantic information which would constitute an algorithm for computing the verification conditions (VC's) of a program with respect to given pre- and post-conditions. This extended grammar can be input to the CSG, which would automatically implement the VC-generating algorithm. Since we want to avoid as much as possible duplicating the large task of writng the front end of a compiler, we may ignore some of the Ada semantic checks and concentrate on the descriptions of the verification conditions to be generated.

Some work which is useful for our purpose has already been published. There is, for example, an attribute grammar description of Ada [UHL 82].

We illustrate the CSG approach by doing an example. We show (approximately) what the input (i.e., an attribute grammar) to CSG would look like for a simple language of while-programs. We first review attribute grammars.

Attribute grammars are context-free grammars with attributes attached to the non-terminals. Associated with each production of the grammar is a semantic rule which defines the values of the attributes. These attributes come in two types: inherited and synthesized. Each production must compute all the synthesized attributes of the unique non-terminal on the left hand side of the production and all the synthesized attributes of the non-terminals on the right hand side of the production. Conversely, this computation can make use of all the inherited attributes of the of the right hand side non-terminals and all the synthesized attribute of the left hand side non-terminal. Considering the parse tree as branching downward, we see that the values of synthesized attributes move up the tree from the leaves to the root while the values of inherited attributes move down the tree from the root to the leaves.

We start with a context free grammar which generates while-programs.

```
        S <== S1; S2

        S <== if b the S1 else S2

        S <= while b inv I do S1 end

        S <== x := e
```

This grammar defines programs S starting with the assignment statement and forming new programs by compositions, conditional and iterations. Note that the iteration or while statement has an invariant I, a boolean expression which is supposed to be assigned the usual meaning – that it remain true after each completion of S1. We now look at some of the productions of the attribute grammar.

In the attribute grammar for while-programs the while-program non-terminal S has inherited attribute PostCond (a predicate) and synthesized attributes VC (a list of VC's) and PreCond (another predicate).

```
        S <== S1; S2          S.VC = S1.VC + S2.VC
                              S.PreCond = S1.PreCond
                              S1.PostCond = S2.PreCond
                              S2.PostCond = S.PostCond
```

This means that S1 and S2 synthesize their PreCond's and VC sets from lower-down in the parse tree while S inherits its PostCond from higher-up. Hence S1.PreCond, S2.PreCond, S1.VC, S2.VC and S.PostCond are assumed available.

The other productions are:

```
S <== x := e                      S.VC = []    -- the empty list
                       -- S.PreCond = Subst e for x in S.PostCond

S <== if b the S1 else S2
                      S.VC = S1.VC + S2.VC
                      S1.PreCond = (b & S1.PreCond) or
                                   (~b & S2.PreCond)
                      S1.PostCond = S.PostCond
                      S2.PostCond = S.PostCond

S <== while b inv I do S1 end
                      S.VC = S1.VC + [(I & ~P) imply S.PostCond]
                      S.PreCond = I
                      S1.PostCond = I
```

The reader familiar with Hoare logic can see that this attribute grammar generates the usual Hoare rules for while-programs.

The syntax-directed editor produced by the CSG can be configured to write the VC's of a program to a file. To the CSG, VC's are just strings, and the VC's can therefore be produced in any desired format. Accordingly, one could choose the format to match the syntax of some existing theorem prover, such as Boyer-Moore, PRL, or LMA. If the theorem prover certifies all the VCs, then the program is correct.

There are several advantages to building near-term verifications systems with the CSG, especially if one wishes to pursue the piecemeal strategy we've outlined:

- All the software exists and the proof technology involved is well understood.
- A byproduct of the approach is a syntax directed editor for asserted Ada.
- Experimenting with proof rules will be quite easy even without using a theorem prover. This is important in developing a proof system that evolves as better and better proof rules for Ada are developed.
- The approach meshes well with longer term solutions that involve the intermediate language DIANA, the specification language ANNA, and various APSE tools being developed for Ada, because attribute grammars are standard compiler technology and form the basis for many language tools.

2.2. Standards for Ada Verification Environments

Questions of standardization are often controversial, and there are strong generic arguments both for and against. Standardization eases problems of communication and co-operation but can also restrict the content (and therefore the interest) of those communications.

Judgments and compromises for the sake of standardization leave their marks everywhere on the design of the Ada language. Ada is itself a standard, but Ada's design takes great care to avoid imposing unduly on implementations, allowing variety and, hopefully, improvement in the design of compilers.

Evidently, standardizing some features of verification systems could cut down on design time for verified systems, on the effort required to develop new tools or to move existing tools to new machines, and on the frustrations of programmers. One counter-argument is that verification environments are not like languages; we have much less practical experience in their design and construction, and it is therefore much less clear what can and should be made standard.

A decision on standards, if any, for verification environments is a long-term project. Certain questions, however, should be considered now, so as not to foreclose future choices:

- Some features of Ada environments are now being standardized. Are there any needs unique to verification systems that are not being adequately addressed by these proposed standards? How would such standards affect the prospects of developing standards for verification systems at some time in the future? Most of this chapter is devoted to that question.

- If there are to be libraries of Ada software, a standard specification language for describing the the libraries' entries will be necessary, and even in the near-term, users of Ada will be adopting one, possibly a homegrown language, willy-nilly. What can we hope for? The obvious near-term answer is ANNA, the only existing formal specification language for Ada.

Chapter 4 will contain a brief discussion on the possibilies of standards for accepting verification environments.

2.2.1. Standard Ada Environments

The Ada language has been defined as a DoD standard (ANSI/MIL-STD-1815A-1983), but Ada language environments have not. In fact, several large Ada environments have been funded and are being developed simultaneously: Ada Integrated Environments (AIE), Ada Language

System (ALS), Worldwide Military Command and Communications (WWMCCS) Information System (WIS). Few functions that they support, other than compilation of Ada programs, will be compatible.

Some efforts have been made toward standard Ada environments. The STONEMAN report [DOD 80] decomposes the construction of environments into layers, the outer-most layer of which contains the greatest functionality and is called an Ada Programming Support Environment (APSE). It is at the APSE level that verification environments will be constructed. A middle layer, called the Minimal APSE, or MAPSE, contains common, less specialized tools such as general purpose text-editors, the machine-independent parts of Ada compilers, and linker/loaders. The inner-most layer, the Kernel APSE (KAPSE), is expected to be transparent to the user, and should contain most if not all of the machine-dependent features. The intent is that the MAPSE and APSE be easily transportable once the KAPSE is retargetted for new hardware.

Naturally, different Ada language environments are being designed with different KAPSE's. It is possible that in the long run competition will create a de facto standard KAPSE; in the mean time, individual tools will not be easily transportable between the various APSE's. One possible solution to this problem may be found in the Common APSE Interface Set (CAIS). The CAIS has been developed to define a minimal standard KAPSE interface, described as Ada package specifications. It was designed to be easily implementable using both the ALS and AIE KAPSE's. The CAIS is a proposed military standard.

2.2.2. The Needs of Verification Environments

Let's first consider the process of verification abstractly. The programmer who wishes to create a working, verified program will follow these steps no matter which verification environment he has chosen:

- Describe in some way the desired result.
- Find related software modules already written, and possibly verified, that can act as components of the final program.
- Invoke the tools of the verification system on the whole program.
- Prove the combined program correct (this can involve varying degrees of human interaction with the verifier).
- Invoke the Ada compiler either directly or indirectly, link modules, load, and

execute. (In general, one expects the compiler to use information acquired during the verification.)

Note: We will regularly and loosely refer to "verified programs", and to "programs" with the "verified" stamp, but of course what is verified is not a program but a program-specification pair.

Standards can ease this process in three broad ways:

- The collection of tools in the verification toolset may be transportable as a unit. It may be desirable to transport a verification environment from one APSE to another APSE with a minimum of modification of the tools. This requires only that the tools' interface to the KAPSE be standard.

- Individual tools in the toolset may be reusable; individual tools from one toolset may be usable with tools in another. A verification environment may consist of several individual tools: theorem checker, theorem prover, verification condition generator (VCG's), etc. "Reusability" of individual tools goes further the transportablity of the whole set. It requires that the interfaces between the verification tools themselves be standard. ("Reusability" in this context is a special case of general "reusability" of Ada software.)

- Different verification environments may be interoperable; they may be able to exchange some of the data on which they work. Most important among this data is the final product: modules of verified code with associated formal specifications. In this case "interoperable" means that verifications done in one verification system can be "understood" and "used" in another. Practically speaking, this requires that the two systems must use the same specification language (or languages intertranslatable on the basis of a common semantics), and the same (or easily intertranslatable) data formats for "stamped" code and specifications.

The following sections discuss these three kinds of standardization.

2.2.3. Transportability: CAIS

Like any APSE, a verification environment will be easily transportable between different machines if it is written in Ada without low-level features, and if the Ada compiler has already been transported to the new machine. Transporting a verification environment to a new APSE will likewise be easy if the verification environment is built from the same standard set of primitives as the target APSE. Is the CAIS suitable as a KAPSE for construction of verification tools?

The CAIS is a collection of Ada packages intended to provide a standard interface between APSE's and their supporting architectures. It therefore promotes the portability of source code, in particular, of software development tools. The CAIS provides, in effect, a small "operating system" interface on which such tools could call in a uniform way.

CAIS models the system entities as nodes (users, files, devices, processes, or "structural nodes") connected by arrows (called "relationships"). The CAIS packages provide such capacities as the ability to create or delete a node, to grant access rights, to move through the graph of arrows from node to node, to spawn (terminate, suspend, etc.) processes, and to carry out some I/O functions.

The current CAIS proposal does not contain standards for:

- Inter-tool interfaces, such as the data formats for the program library, the text format in editors, etc.
- Interoperability, such as standard external data representations for transferring data from one environment to another.
- Facilities for archiving.

Our concern here is only with the question: are there needs unique to verification tools that are not well met by the existing proposal for CAIS?

The proposed CAIS requires a verification system to treat source files, specification files, proof history files, object files, executable files, and any other files intermediate in the verification process, as nodes in the CAIS tree, and to label their relationships via user-defined attributes attached to nodes. For example, the special attribute "verified" would be attached to object-file and executable-file nodes that had been successfully passed through the verifier. Files containing the formal specifications must also be associated with their respective code files. The CAIS facilities for node relationships are sufficient for this structure, so our ability to build a verifier on top of the CAIS seems certain.

However, since the attribute "verified" is supposed to assure the user that software so stamped has some degree of correctness, we must consider the CAIS provisions for ensuring integrity and access control. The following things must be guaranteed:

- The stamp 'verified' can be placed on code files only by known and approved verifiers. This is a straightforward use of access control for processes. Code files might be copied and the copies modified, but such copies cannot inherit the 'verified' attribute.
- Writing or modifying a code file removes the 'verified' attribute unless done by

certain trusted processes, meaning preserving transformations (such as stripping the comments from the file). This prevents direct tampering with verified code. What must be maintained invariant is the relation between code and its specifications. Accordingly, if the specifications of a module are modified it must be guaranteed that all logically dependent modules will lose their "verified" stamp (unless, again, the modification is done by a a trusted process). This question of "logical integrity" seems a little subtle, and is not addressed by any of the discussions of integrity known to us.

- The approved verifiers themselves, and files holding partially completed proofs, must be protected against modification. This is a straightforward use of mandatory access control for integrity.

CAIS specifies an access control interface, but notes that its mechanisms for access control are only recommendations, and may be replaced with the stipulation that the "semantics" of all other CAIS interfaces are implemented as specified. The above are minimum requirements for the access control underlying a verification APSE.

All verification systems must face the awkward question "Who guards the guardians?" Ada verification tools will themselves be programs (presumably Ada programs), and will rely on the correctness of the packages in CAIS or some standard interface. Therefore, : complete specification of a standard KAPSE interface will not truly be adequate without formal specification of the semantics of that interface. A formal specification would allow a formal proof that some KAPSE implementation is correct and can support the integrity properties given above. The semantics of the proposed CAIS interface are given only informally (and, because of the way they are stated, cannot be formalized without a formal semantics for the whole Ada language).

2.2.4. Reusability

A verification environment will consist of several tools. Can tools from one toolset be used in another? There is some profit for tool developers, and in the long run for the tool users, in the interchangability of parts. The most obvious example is standardizing the theorem-prover interface: agreeing that theorems supplied to the standard interface fall into some particular formalization, e.g., many sorted first-order logic over Ada types. Then advances in theorem-prover technology could proceed independently of development of VCG's, etc., and systems could be altered to exploit the best technology available.

However, the counter-argument demands flexibility. The field of formal verification and tools to support it is dynamic and changing. There has been and continues to be much work on

VCG's, theorem-provers, decision procedures, etc. Consider the two main approaches to verification:

- The direct approach: the programmer writes both code and formal specifications, verification conditions (VC's) are generated and submitted to a theorem prover.

- The transformational approach: the programmer constructs a specification in some formal language, and then invokes a sequence of meaning-preserving tranformations transforming that specification into a program.

If a programmer in a transformational system succeeds in fully implementing his specification with a sequence of system-supplied transformations, then that sequence itself is a verification of his program. He may, however, have to supply transformations of his own devising and prove that they are meaning-preserving. A transformational system's theorem-prover might not be the sort of prover and checker of specification-language assertions that would be usable in a verification system based on generating VC's. It might operate on an idiosyncratic extension of a fragment of the specification language.

It is far too soon to know which of the competing approaches will prove superior, and therefore too soon to know which kinds of tools should be standard, much less what their interfaces should be.

2.2.4.1. Special Ada Compiler Interfaces: DIANA

Interactions between verification tools and the Ada compiler are a special case of interest. Verification tools may interact with the Ada compiler for several reasons:

- Tools may take output from the compiler, such as the symbol table for a program, to avoid the necessity of reparsing the program in the verification environment.

- Tools may use the proof of a program to provide input to the compiler, to help optimize the compilation on the basis of extra knowledge of the algorithm gained from the verification process. They would, in effect, be automatically-supplied pragmas.

A program P might call a procedure Q with exceptions E1, ..., En. The formal specification of Q would include some specification of the conditions under which each exception was raised. If it can be proved in the verification environment that the conditions under which, say, E1 is raised never occur in P, then the compiler need not generate the code for P's exception handler for Q. This is an example of space optimization.

An example of optimization in time: if it can be proved that within a module P, array index checking, range constraints, or other checks will necessarily be met, then a SUPPRESS_CHECK pragma can be supplied automatically for that type of check within P. It might perhaps be more prudent to say that if one is forced to suppress some run-time chec: s, the verifier could help decide which checks could be removed with the least risk.

Since verification tools can profitably interact with the Ada compiler, perhaps a standard, compact description of Ada can serve as the medium for interaction. Can DIANA serve this function? DIANA is an intermediate language which encodes a "lexical, syntactic, and static analysis" of Ada programs, but not the results of "dynamic semantic analysis, of optimization, or of code generation". It is intended to be used as the input to pretty-printers, syntax-directed editors, compilers, etc.

The basic principle is straightforward, encoding the program text as a tree, but certain features of Ada, such as separate compilation, complicate the picture. A DIANA representation is officially defined as an abstract data type, consisting of the "syntax tree", various "attributes" attached to the nodes of the tree, and various tree manipulations. For example, the attribute "semantic value" is defined on certain expressions which can be evaluated statically, and returns the value of such expression.

A program is a DIANA producer if, given an Ada program, it produces an output which is, essentially, a superstructure of the DIANA tree associated with that program. It must contain at least as much information as a DIANA tree and express that information in proper DIANA form. For example, the output must give the correct values for all the DIANA attributes.

A DIANA consumer is a process which, in producing its output, must not rely on any more information than is contained in the standard DIANA tree (and in the standard form) – although it may have the capacity to make use of extra information, should it be present. For example, a compiler would not be a DIANA consumer if it had to be provided with the values of certain non-static expressions.

The original program text is recoverable from a Diana tree, except for:

- Trivial normalizations (e.g., every statement of the form "begin block B ... <body> ... end block B;" might be reconstructable only as "begin ... <body> ... end;")

- Comments (the value of the "comments" attribute is not defined, and indeed the attribute need not even be suported by a DIANA producer or consumer).

A verification system will surely not be a DIANA consumer, since it will surely rely on the presence of more information than is contained in the Diana tree. For example, a Hoare-style system would operate on programs containing "embedded assertions", possibly in the form of attached comments. On the other hand, there seems no reason why the front-end of a verification system should not be a DIANA producer.

2.2.5. Interoperability: Standard Specification Language

There are good reasons to work toward the goal of a standardized specification language for Ada. One can imagine an ideal world in which there are libraries of verified Ada programs, each possessing a specification which describes the essence of its program's function in a formal specification language. Programmers in this ideal world use the formal specifications of programs they intend to build as keys, aided by some form of automated low-level reasoning, to search through the catalogue of formal specifications of library items for useful program units. Having built a completed program from these verified units, a programmer could be confident that his program would be verifiable solely on the basis of the formal specifications of library components.

If different libraries were based on different specification languages, they would be unable to communicate with each other, even though the library packages themselves are fully reusable at the level of Ada syntax.

At present, the only specification language for Ada that has been developed to any degree is ANNA. It does not cover all the features of Ada; in particular, it does not cover concurrency, and is, by deliberate design, as conservative an extension of Ada as possible.

Even specification languages that cover all Ada constructs will differ in the kinds of assertions they make about programs using those constructs.

One might get general agreement that the language should contain a many-sorted, first-order logic ov.. Ada's types. There would be little agreement on whether the logic should be classical or constructive, what additional type-constructors to include, how to specify concurrent processes, or how to specify real-time processes.

2.3 References

[DOD 80] U.S. Department of Defense, Requirements for Ada Programming Support

Environments, February 1980.

[LISKOV 77] LISKOV, B., et al, "Abstraction Mechanism in CLU," Communications of the ACM (August 1977).

[REPS 84] Reps, T. and Alpern, B., "Interactive Proof Checking," Conference Record of the Eleventh Annual Syposium on Principles of Programming Languages (1984): 36-45.

[TEITLBAUM 81] Teitlebaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Communications of the ACM 24,9 (September 1981): 563-573.

[TEITLBAUM 81] Teitlebaum, T. and Reps, T., "The Synthesizer Generator Reference Manual," Cornell Computer Science Technical Report 84-619, 1984.

[UHL 82] Uhl, J., et al, An Attribute Grammar for the Semantic Analyses of Ada, Lecture Notes in Computer Science 139, Springer Verlag, NY, 1982.

3. SPECIFICATION LANGUAGES

3.1. Introduction

A primary goal influencing the design of Ada was the inclusion of features supporting modularity and the development of reusable software. Ada program units contain a specification layer textually distinct from their bodies. By "specification" Ada means syntactic information sufficient to use or call the specified unit. Aside from informal comments Ada provides no way to specify a unit's functionality. To enhance support for reliable, maintainable, reusable software there is a need to develop an Ada formal specification language. This formal specification language would be used to:

- Present the semantic interfaces between Ada program units. Such an interface would communicate to users the total effects of calling such units; such effects would include:

  - The results of normal execution.

  - The conditions under which named exceptions are propagated out of the unit.

  - The conditions under which predefined Ada exceptions are handled within the unit and the effects of such handlers.

  - The specification of the unit's concurrency features, e.g., the conditions under which rendezvous occur and what are the effects.

  - The effects of elaboration and package initialization. (These are non-trivial; an interesting exercise in the Dear Ada column in the July-August 1983 edition of The Ada Letters shows what package elaboration can lead to.)

- Encourage the use of generics by providing a way of semantically restricting generic parameters. For example, the generic parameters to a generic tree_sort package might include a user supplied type and a user supplied linear order over that type. At present only the type signature of the latter function can be specified. One would like to add to this a semantic specification which states that a linear order is needed. This would communicate that the specified effects of the package's subprograms can be expected only if the user supplied function is indeed a linear order.

- Encourage the use of Ada as a Program Development Language; formal specifications would play the role of informal pseudo-code. The use of formal specifications in program development is well illustrated in [GUTTAG 80].

- Support formal design verification, that is, proofs ·hat a design entails certain system properties. Such proofs can be done prior to implementation.

- Support program verification, software reusability must ultimately rely on assurance that code satisfies its specifications. Formal program verification, when technologically feasible, provides a strong assurance criterion. In the case of concurrent programming, where testing is non-repeatable and inconclusive because of asynchronous events, formal verification is perhaps the only way of establishing confidence in code. Verification is emerging as an important means of ascertaining correctness of concurrent programs (there is a large literature in this area, e.g.[OWICKI 82], [PNEULI 80]).

- Support the generation of run-time monitors. In the absence of formal verification constructively interpretable specifications can be mechanically transformed into code which performs run-time checks to determine whether the specifications are being violated. Ada range constraints in subtype declarations are a simple example of a formal specification being compiled into a run-time check. In this case an exception is raised when the specification is violated. For a more sophisticated example see [GERMAN 82].

- Support rapid prototyping. Several formal specification languages support transformations of formal specifications into inefficient, but correct implementations. Such rapid prototypes have many uses in the software development process. To support rapid prototyping it is not always necessary to restrict specification language constructs to what is ordinarily thought of as constructive. Experience with interpreting and compiling very high level programming languages like SETL [DEWAR 79] has shown that if efficiency is not a requirement then higher order constructs can be systematically replaced by executable text.

3.2. ANNA

The fundamental limitation of the most advanced program verification system, the Gypsy Verification Environment developed by Don Good at the University of Texas [GOOD 78], is the lack of expressive power in its specification language. This limits what can be stated and proved about Gypsy programs.

We will sketch some features of an imaginary specification language for Ada, here called SPEC. (SPEC is not an acronym but an actual name.) We will use Gypsy and Anna as strawmen and sketch SPEC as a collection of improvements to Anna. The ANNA annotation language developed at Stanford University under the direction of David Luckham is the most widely known formal specification language for Ada. The latest publicly released document is [ANNA 84], which we refer to as RMA. It unfortunately omits several important Ada features (concurrency annotations, for example). The original description of ANNA is contained in the paper [KRIEG 80] and an example of its usage is presented in [KRIEG 83] which we refer to as CC.

ANNA is described by its designers as a "cautious extension" of Ada and SPEC is a cautious improvement to ANNA, and as such, perhaps a mid-term rather than a long-term goal. We view SPEC as an extension of Anna which encapsulates and hides some of the latter's logical freedom.

Since ANNA is the only Ada formal specification language project which has generated documentation we shall describe it in some detail. As we shall see, Anna restricts itself to expressing specifications in first order logic, using Ada entities as terms and predicates within this logic. We shall argue that there is also a need for higher order constructs in a specification language where intelligibility is the key requirement. As people in the Ada community are aware, the NYU Ada/Ed compiler was the first to be validated. It contains only 9,000 executable SETL lines (and 5,000 SETL comments) and was produced in nine person years! SETL supports the use of entities from the universe of set theory. Ada/Ed is ultra-slow and its generated object code is not efficient, but that is a secondary issue in a rapid prototype. Essentially, SETL permits a high order description of an Ada compiler to be itself compiled into an implementation.

ANNA programs are Ada programs with formal comments. Anna formal comments can occur anywhere in Ada text where informal comments can occur. They consist of virtual Ada text preceded by –: and annotations preceded by –!. The virtual Ada text consists of Ada declarations, statements, etc. which would be legal Ada if the –: symbol where removed. The Ada text outside the virtual text is called real Ada text for distinction. No real Ada object can be changed in virtual Ada text and no virtual Ada entity can be referenced in the real portion. The objects and program units introduced and manipulated in the virtual Ada text are used in the annotations. They are specification artifacts.

Annotations are written in first-order logic, that is, the Ada syntax of Boolean expressions is extended to include quantification ("for all" and "there exists") over Ada subtypes. The unquantified variables of the annotation are virtual or real Ada variables visible at the point where the annotation occurs. Annotations should be distinguished from the more familiar assertions which one finds in program verification environments such as Gypsy. Assertions and annotations have the same syntax, but they differ in semantics. A program is consistent with its assertions if, during every execution, whenever control reaches an assertion the latter is true in the current state. Two special kinds of assertions are subprogram pre- and post-conditions. The former are attached to the entry point of the subprogram and the latter to the exit points.

In contrast to assertions, ANNA annotations have a scope determined from their position in the text according to the Ada scope rules. Annotations are to be true throughout their scope and not only where they first occur. They generalize the Ada concept of constraint: they constrain the values of the program variables which occur within them. The scope of the annotation determines where the constraint must hold. Ada text is consistent with its annotations if in all executions the computation states satisfy the constraints imposed by the annotations. Assertions are special cases of annotations, namely, annotations whose scope is a single control point. For example, an annotation prefixed by the reserved word out is an assertion.

Let us consider some examples. Suppose one wished to define a subtype of INTEGER consisting of all non-zero integers from -N to +N. The range constraint -N .. N is a legal Ada subtype indication but there is no way to indicate the removal of 0 from the range. Using ANNA this could be done by

```
        subtype SYMMETRIC is INTEGER range -N .. N;
        --! where S:SYMMETRIC => S /= 0;.
```

Because of its location the annotation has the same scope as the subtype mark SYMMETRIC. Thus Ada text containing this annotation would be consistent with it if the condition /= 0 were also to be true whenever a range constraint check is made on an expression to determine whether its value is SYMMETRIC.

Some further Anna -

```
        --:function IS_PRIME(P:NATURAL) return BOOLEAN;
                --!return not exist X, Y :NATURAL range
                --!    2 .. NATURAL'LAST =>  X*Y = P;
subtype PRIME is NATURAL;
        --!where P : PRIME => IS_PRIME(P);.
```

Here an annotation is being used to define the subtype PRIME. To do so the virtual function IS_PRIME is used. This function does not have a body but does have its own annotation, which is a post-condition (i.e. an assertion) which asserts that its output is the usual definition of primeness.

Neither functions with side effects nor procedures can appear in annotations. In order to permit procedures to be mentioned in annotations, ANNA adds to Ada a predefined attribute OUT with the semantics that, for any procedure P, P'OUT is a function returning a record containing the final values of the out and in out parameters of P after a call. The formal parameter names of P are the component names of this record. This is an interesting idea.

Gypsy would benefit substantially from such an addition.

While the RMA presents ANNA's semantics informally, its real semantics is determined by transforming ANNA text into asserted Ada, thus reducing ANNA semantics to Ada semantics and the meaning of assertions. This reduction of annotations to assertions is mentioned in ARM but not described. CC, on the other hand, presents some examples.

Assertions have been used classically to describe the pre- and post-conditions of subprograms. ANNA adopts this usage, and extends it to generic subprograms and packages. Package annotations can make use of the ANNA attributes TYPE, INITIAL, and STATE. If X is a package then X'TYPE is the type of the states of X. It behaves as a limited private type exported from X. Outside of X no structure is visible, and X'TYPE is treated inside X as a record whose components are all the local objects and packages in the declarative region of X. X'INITIAL is the initial value of the state of X after elaboration and X'STATE is the current value.

3.3. The Underlying Logic

Our fundamental requirement on SPEC is that it supports program verification. This entails that all specification language constructs should be mappable into properties of the purported Ada implementation which we know how to verify. This is a non-trivial requirement since much research remains to be done on Ada verification, particularly in the area of tasking. The design of SPEC ought to be coordinated with this research. Original HDM, for example, contained specification language constructs which were not verifiable and had to be dropped from later versions.

To support such verification, there must be a formal semantics relating Ada and SPEC. ANNA attempts to avoid this issue by mapping ANNA informal semantics directly into informal Ada semantics. At present this is the only reasonable thing to do. It is done by replacing annotations by embedded assertions at key places within the the annotation's scope. Unfortunately, the mapping presented is not always correct. In any event the problem of formal Ada semantics is only deferred, and not avoided.

The meaning of embedded assertions in a program presents difficulties which must be faced. The discussions in both RMA and CC are inadequate from a logical point of view. Assertions may not have a defined value in a given computation state for several reasons.

- Program variables (real or virtual) occuring in the assertion may not be defined in the current state (Class I).

- Evaluation of user-defined functions occuring in the assertion might not terminate (Class II).

- Evaluation of user-defined functions occuring in the assertion might raise an exception (Class III).

- User-defined functions might not have a body but only a post-condition (as in the case IS_PRIME above) and the latter might not determine the function value uniquely (Class IV).

- User-defined functions might not have a body but only a post-condition which is inconsistent, no value can satisfy it. (Class V).

It is interesting to compare Ada's and ANNA's to Gypsy's handling of these problems. In Gypsy, all types have default initial values which can be overwritten when a variable is declared. Thus a Type I difficulty can not occur. In Ada, types other than record types do not have default values and the problem of reading an unitialized or undefined variable is particularly thorny. There is no predefined Ada exception corresponding to this case. The Ada standard calls programs which read undefined objects "erroneous". Erroneous programs give unpredictable results and there is no requirement that an erroneous error be caught at either compile time or run time. For this reason, undefined variables cannot be checked by user-supplied program text in order to raise a user-defined exception. If the variable is really undefined, the checking program will be erroneous and thus unpredictable.

ANNA, to its credit, adds two Boolean valued predefined attributes, DEFINED and COMPLETE. If X is a scalar variable, then X'DEFINED returns TRUE if X has a defined value. If X is a composite variable, then X'COMPLETE returns TRUE if all scalar components are DEFINED and all composite components are COMPLETE. Good ANNA style would suggest that annotations should always use these attributes to check the program variables which occur within them. If a FALSE occurs, the ANNA predefined exception, ANNA_ERROR, should be raised. In this way Type I difficulties in the above list could be reduced to Type III. Unfortunately, this discipline is not enforced in ANNA. To preserve the soundness of the underlying logic, such checks should not be left to the specifier's discretion and wouldn't be in SPEC.

It should be mentioned that the reason Ada allows un-itialized variables to be read by program text is that program variables can be "wired" to hardware addresses (e.g., I/O data and control registers), Ada's low level features. In these cases the variables are written from outside the High Order Language (HOL) level. On the other hand, it is clear from the program text which variables are so wired and from a logical point of view they can be considered to be initialized. This remark shows why so-called erroneous programs are useful; programs which read hardware registers will be unpredictable at the HOL level.

Gypsy essentially ignores the possibility of Type II errors. The underlying theorem prover assumes all expressions are defined. The rationale is that this comes under the meaning of partial correctness as that term is understood in program verification. A program is said to be partially correct with respect to its pre- and post-conditions if whenever actual parameters satisfy the former and the program terminates then resulting values of the actuals satisfy the latter. As usually understood, partial correctness makes no mention of the possibility that the calculation of the pre- or post-condition (or any intermediate assertion) might itself not terminate. This is because in textbook program verification, the functions and predicates appearing in assertions are viewed as mathematical functions, not user-supplied subprograms. Extension of the term "partial correctness" as is done in Gypsy and ANNA leads to logical difficulties that remain to be thoroughly explored in the research literature.

We suggest that it would be possible, in SPEC, to avoid type II difficulties. This can be done by restricting the control constructs which could appear in the bodies of virtual functions mentioned in assertions. A classic result of Meyer and Ritchie guarantees that every primitive recursive function of natural numbers can be defined when loop statements are restricted to indexed loops (with or without additional exit statements). Experience has shown that such functions are adequate for specification purposes. Of course, real Ada functions might appear in assertions and their bodies cannot be restricted. In this case an appeal to the partial correctness notion does not appear so self-serving. The virtual functions are introduced for specification purposes only; they should be mathematical functions. The real functions which appear in the assertions might very well not terminate but partial correctness always makes the assumption that the executable objects terminate.

Type III difficulties are essentially ignored in both Gypsy and Anna by appealing to that old flag of convenience, partial correctness. Our proposal that virtual or specification functions be total cannot be extended to avoid Type III difficulties. There is no way to guarantee that predefined exceptions not be raised since their semantics are in some cases implementation determined. Critics of Ada have focused on the Ada programmer's power to handle predefined exceptions as a major stumbling block to the development of reliable programs. This issue must be attacked head on in Ada specification/verification work. The underlying logic must be extended to include exception raising. In this logic the definition of satisfaction of a formula might contain clauses dealing with the cases when mathematical functions and predicates within the formula raised exceptions on certain values. The result would be the replacement of the formula whose truth is being determined by another formula. The main difficulty in working out this logic seems to be dealing with order dependencies: if several exceptions can be raised, which will be?

Most machine instruction set architectures do not determine which exception will be actually raised when several are possible. Suppose, for example, that an instruction is fetched whose op code requires some kind of privileged mode to execute and whose operands map incorrectly under the hardware virtual map to some segment marked no access. Both an illegal_privilege and an illegal_address exception can be raised and the Instruction Set Architecture doesn't determine which. In many cases there is no general policy, different circumstances giving rise to different results. The first exception encountered is raised but which [illegible] encountered first differs from instance to instance. Such an asynchrony could occur in a pipeline instruction decode situation. Since, for efficiency, predefined Ada exceptions would be implemented using these hardware exceptions there can be no guarantee which Ada exception is raised. These gruesome details are reflected by the fact that most Ada expressions are evaluated in some order not determined by the Standard. Hence, which exception is raised is not determined.

The only viable approach which includes exceptions is to treat Ada as a non-deterministic language; given inputs give rise to several computation sequences all of which must satisfy the annotations. A formula in the underlying logic of SPEC might evaluate in a given state to TRUE, FALSE, or a new set of formulas indexed by the names of exceptions. Thus, evaluation of assertions could lead to branching which would be coupled with the branching computation sequences correponding to non-deterministic execution.

We now turn to the remaining types of difficulties outlined above. Type IV and V were the cases which could arise in the axiomatic approach to specification. The specification functions and predicates which occur in assertions are given only by post-conditions which either might not determine the value uniquely (Type IV) or might not have any solution (Type V). In Gypsy, the post-condition of a function with return type T is a boolean expression involving the formal parameters and the indentifier "result" which is of type T. It could be a partial specification of the form "result < 0" or an inconsistent specification like "result < result". The former will not lead to any logical difficulties; the latter will. In ANNA, the postcondition of a function with return type T can be presented in three ways. The first is of the form "return e" where e is an expression of type T (see IS_PRIME above). This avoids both Type IV and Type V difficulties. The second is of the form

return t : T => B(t)

where B is a Boolean expression. This is equivalent to Gypsy's

B(result)

and similarly does not lead to logical difficulties in Type IV but does in Type V. The final ANNA form is

return that t : T => B(t)

which implies that t is unique. If B does not have a unique solution then both Type IV and Type V lead to logical inconsistencies.

Just as we restricted the user-defined specification functions and predicates to terminating entities, by restricting the control constructs which can occur in their bodies, we wish to restrict the use of formal specifications so that inconsistent specs can not be written. This would be essential if SPEC were to support rapid prototyping. We are thus led to the following suggestion. Of the three ANNA forms, we will forbid the third, which introduces too many difficulties. If a specification function or predicate without a body is introduced using the second ANNA form then it must be accompanied by a proof of

exist t : T => B(t)

that is, there is an object which satisfies t. Since the formal parameters of the function can also appear in B, t depends on them. The proof can be left pending but ultimately a complete specification would include such a proof. The proof would be in a formal logic checkable by SPEC support tools. Giving a proof that such a t exists is not as difficult as proving that a program for constructing t is correct (i.e., program verification). This is essentially the difference between classical mathematics and constructive mathematics. The former is easier. Non-constructive existence proofs need not explicitly display the object shown to exist. This is particularly true of proofs by contradiction, where the assumption that no t satisfying B(t) is shown to lead to a falsehood.

We are not asserting that the above can be done by magic. Verification of Ada programs (i.e., programs which really make use of Ada features and are not just sub-Pascal programs) is not yet feasible since many questions of Ada semantics remain open.

It seems reasonable to attempt to build an environment for Ada "design verification" first. In design verification one shows that the overall design satisfies its specification by assuming that the pieces entering into the design satisfy their specifications. Such proofs are meant to be undertaken in the absence of the pieces' bodies. The danger is that the whole undertaking might be undermined because the specifications supplied are inconsistent. If and when provably correct bodies are supplied doubts will be removed. But design verification should be performable before implementation.

SPEC is a middle way. As part of design verification we demand proofs of consistency of specifications. Such proofs can be given non-constructively in a non-algorithmic manner. Such proofs will compile but they can hardly be called programs. Nevertheless they will generate a rapid prototype.

Let us illustrate the above. We can assert as a post-condition of a sort function which operates on unconstrained integer arrays that the resulting array is a permutation of the input in increasing order. First note that the natural way of writing this post-condition uses a quantification over functions (i.e., there exists a permutation, etc.). ANNA does not support such an expression since quantification must be over Ada types and function spaces are not Ada types. We return to this point in the next section. How can one prove such a permutation exists? One can form a quick proof using SETL-style constructs in the following manner: assume some integer array can not be sorted, chose one such of shortest length. Its set of range values, being finite has a smallest element. Form a new array with this smallest component missing and, since its length is smaller than the original, it can be sorted. Choose a sort of the new array and tack the formerly chosen smallest element on the left. The result sorts the original array. A contradiction: this proof can be transformed into a recursive SETL program, but even without doing so an ultra-inefficient prototype can be found by replacing the existential quantifier in the post-condition by an "or" and systematically enumerating all permuations of the original array until a sort is found.

3.4. Higher Types

ANNA annotations and assertions are built from relational operators applied to virtual Ada expressions using the Boolean operators and quantifiers over Ada subtypes. Since Ada does not provide a power set type-constructor the expressive power of the annotations are restricted. A typical use of set theory would be an annotation for a recursive data type like trees. Consider the following:

```
generic
        type elem is private;

package trees is
        type tree is private;
        nil : constant tree;
        function join(t1, t2:tree; e:elem) return tree;

        private
                type treenode;
                type tree is access treenode;
                type treenode is record
                        tip:elem;
                        left, right:tree;
                end record;
                nil : constant := null;

end

package body trees is
        function join(t1, t2:tree; e:elem) return tree is
                t3 : tree;
                begin
                        t3 := new treenode;
                        t3.tip := e;
                        t3.left := t1;
                        t3.right := t2;
                end join;
end trees
```

What this package defines is not really trees, since there are objects of type tree which aren't trees (e.g., an access object denoting a treenode which has itself as its left component). Real trees form a subtype

```
        subtype real_tree is tree;
```

which requires the annotation which declares that nil is a real_tree, that join(t1, t2, e) is a real_tree if t1 and t2 are and furthermore that any subset S of the type tree which contains nil and join(t1, t2, e) for all e : elem and for all t1, t2 : in S contains all real trees. This annotation requires a quantifier over the power set of the type tree. Similar annotations are needed for other recursively defined types; they are necessary for supporting proofs over these types. Trees as defined above should actually be a limited type. The definition of "=" on trees would be recursive and would not terminate in many cases in which its arguments were not of subtype real_tree. This would conflict with our previous proposal that all functions appearing in annotations be total ("=" would no doubt appear in an annotation).

Before one can add sets to the assertional part of ANNA, one would need to develop a theory of sets consistent with Ada's typing philosophy. Empty subsets of distinct types should be considered distinct; in ordinary set theory they are equal. On the other hand one would like to overload the symbol for the empty set as well as all set operators. The set theoretic constructions should be chosen to support effective realization, as in SETL, in order to yield rapid prototypes.

In addition to the set type-constructor, one has need of higher order function space constructors at the specification level. We saw an example in the previous section where the post-condition for a sort function needed to quantify over permutations. Subprograms with generic subprogram parameters are actually functionals mapping function into functions. While Ada itself does not support further iteration, allowing iteration at the specification level sometimes simplifies descriptions. The instantiation primitive "new" takes a generic subprogram and an actual program and produces a new program. Thus it itself takes a functional and a function and produces a function, making it a function of a functional and thus of type 3.

Extensions to Ada have been proposed which do allow various higher order iterations of Ada features. The Intel 432 chip [INTEL 83] for example, allows packages to be types. Objects of package type can be declared and package bodies can be assigned to them as values. This is a powerful abstraction device [BUZZARD 18]. Other recommendations for extending Ada to allow greater abstraction are given in [BOUTE 80], [JESSOP 82] and [WEGNER 83]. The objection to including these extensions in th language itself is their lack of efficiency without special architectures to support them, although from the point of view of formal specifications many of these proposed Ada-like abstraction mechanism are very appropriate.

As Wegner makes clear in his article previously, Ada does not allow subprogram and package program units to be first class objects. A first class object has a type, can be declared, can be passed and returned as an actual parameter, and can be assigned to. Ada is unsystematic since tasks, which are also program units, can have a type and be declared and passed as actual parameters. He suggest removing all distinctions among the various kinds of Ada "entities" to yield type completeness in the senf of [DEMERS 80]. This "completion" of Ada's typing philosophy corresponds to the use of an iterated power set operator in the assertion language. Essentially, whenever we have entities of types A1, ..., Ak and B, we would like entities of type [A1 x A2 x ... x Ak -> B], which are operations with inputs of type A1, ..., Ak and output of type B – and we want these entities to be first class. These operations can have side-effects (something not allowed in Gypsy, but allowed in HDM and

Ina Jo) and these side-effects are dealt with by considering the operations as having an extra input and output argument of type "state". This state parameter is implicit at the specification level and can be made visible only by a deliberate act of the specifier. At the proof level the system explicitly adds the state parameter. Ada's visibility rules allow one to control the size and complexity of the implicit state parameter (if it were not controlled, proofs would become enormous). The state is essentially a record indexed by all the variables and packages visible at the point of declaration of the function. The value of the state consists of the variables and the current states of the packages (where package states are defined as in ANNA). A similar reduction of subprogramns to functions occurs in Revised Special. One of the main problems with specifying concurrent programs is the difficulty in defining "state".

3.5. Encapsulating Quantifiers

An important goal of SPEC is that it should be understandable by software engineers possessing only a minimum of extra training in formal methods. Most software engineers feel uncomfortable in the presence of formal logic. While they are familiar with some uses of quantifiers, they have usually not encountered the free, unrestricted usages allowable in most specification languages. Instead of raw quantification, SPEC might use constructs familiar to software engineers, which can be mechanically expanded into an internal logical form. Presently available specification languages, such as ANNA, Gypsy, Ina Jo, SPECIAL, and AFFIRM force the specifier to use first order quantifier logic directly. In contrast, we suggest viewing logic as the "compiled" form of specifications and attempting to provide constructs which encapsulate the use of quantifiers. We believe that all uses of quantification in specifications can be avoided by proper choice of constructs. This is analogous to the elimination of most "gotos" using whiles and repeats.

As an example of the above, consider the following Ina Jo expression, which occurs frequently:

```
A"i:INTEGER (i = j => N"array(i) =
                      x <> N"array(i) = array(i))
```

which is quite cryptic. The

(C => D <> E)

is Ina Jo's IF C THEN D ELSE E. The N" operator is Ina Jo's new value operator. The above specification says that the new array differs from the old array only at index j. In this case the new array's value is x. Compare this to the notation

```
array := array with (array(j) := x)
```

whose meaning would be immediately obvious to any programmer. Furthermore, the latter can easily be mechanically expanded into the former. Gypsy uses essentialy this formalism, called "modified" expressions, for structured types.

The non-procedurality of formal specifications is frequently mentioned as an impediment to understanding specification, since the effects of a state transition are considered as happening not in any order but all at once. Programmers familiar with multiple assignment statements can easily master this form of expression. The Ina Jo expression

```
A"i:INTEGER (i = j => N"array(i) =
                      x <> N"array(i) = array(i))
& N"j = j + 1 & N"x = x - 1
```

could be written

```
array, j, x :=
        array with (array(j) := x), j + 1, x - 1
```

with complete preservation of semantics.

Multiple assignment statements would appear to be inadequate for non-deterministic specifications. Expressions such as

N”v > v

appear often in formal specifications. What's being stated is just that the transform increases the value of the variable v. This can be expressed in assignment form by

v := y where y > v

which uses a new variable y and a "where" qualifier. Such a construct replaces existential quantification, which is mysterious to most non-logicians. The logical translator would rewrite the above assignment as

E”y (N”v = y & y > v)

but the human specifier need not do it himself.

3.6 References

[GUTTAG 80] Guttag, J. and Horning, J., "Formal Specification as a Design Tool," POPL 1980.

[OWICKI 82] Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent

Programs," ACM TOPLAS, 4 July 1982.

[PNEULI 80] Pneuli, A., "On the Temporal Analysis of Fairness," Proceedings of the Seventh POPL, January 1982.

[GERMAN 82] German, S., et al, "Monitoring for Deadlock in Ada Tasking," Proceedings of the AdaTEC Conference on Ada, ACM, October 1982.

[GOOD 78] Good, D., et al, Report on the Language Gypsy, University of Texas, September 1978.

[DEWAR 79] Dewar, R., The SETL Programming Language, NYU, 1979.

[ANNA 84] Stanford University, Reference Manual for ANNA: A Language for Annotating Ada Programs, July 1984.

[KRIEG 83] Krieg-Bruckner, B. and Luckham, D., "ANNA: Towards a Language for Annotating Ada Programs." SIGPLAN Notices 15 (1980): 128-138.

[KRIEG 83] Krieg-Bruckner, B., "Consistency Checking on Ada and ANNA: A Transformational Approach," The Ada Letters 3,2 September 1983.

[INTEL 83] Intel Corporation, iAPX 432 General Data Processor Architecture Ref. Manual, Rev. 3, 171860-003, Sant Clara, CA, 1983.

[BUZZARD 85] Buzard, G. and Mudge, T., "Object-Based Computing and the Ada Programming Language," IEEE Computer 18,3 March 1985.

[BOUTE 80] Boute, R., "Simplifying Ada by Removing Limitations," SIGPLAN Notices, February 1980.

[JESSOP 82] Jessop, W., "Ada Packages and Distributed Systems," SIGPLAN Notices 17,2 February 1982.

[WEGNER 83] Wegner, P., "On the Unification of Data and Program Abstraction in Ada," 10th Annual POPL, January 1983.

[DEMERS 80] Demers, A. and Donahue, J., "Type Completeness as a Language Principle", 7th

POPL, 1980.

4. Far-Term Efforts

This chapter discusses two long-term goals for Ada verification, goals possibly attainable within 10 years: a full formal definition of Ada and standards for accepting verification environments.

4.1. Formal Semantics: The EEC Effort

A formal semantics must underly any serious long-term system for specification and verification. It seems unlikely that the whole Ada language will ever be verifiable, and it would be sufficient for the purposes of verification to have a formal definition only of the fragment of the language dealt with by the verification system. A formal definition of the whole is nonetheless desirable.

The first attempt to define the Ada language formally was the INRIA project [DONZEAU 80]. It was based on an early version of Ada and omitted tasking entirely. [BJORNER 80] was directed toward what one of the editors calls a "pseudo-formal" definition, also of an early version of the language.

The 1983 ESPRIT study [ESPRIT 83] of research centers in Europe, the US, and Japan discovered only two groups working explicitly on the semantics of Ada: INRIA and DDC. The final report of that study consists of two volumes: a survey of then-current research work on formal methods, carried out by Standard Telephone Laboratories (UK); a study, by the Dansk Datamatik Center (DDC) (Denmark), devoted specifically to the Vienna Definition Method (VDM).

To the best of our knowledge there now exists only one long-term project for the formal definition of Ada, variously called "The Draft Formal Definition of ANSI/MI_AL-STD 1815A Ada" or the "EEC-Ada formal definition project." It is sponsored by the Commission of the European Communities, under the Multi-Annual Programme in the Field of Data Processing, and is a joint project involving: the DDC, and, in Italy, the Consortium for Research and Applications in Informatics (CRAI) and Istituto Elaborazione Informazione (IEI). It is intended that this project will result in a formal definition of the Ada language helpful to many Ada users (language designers, implementors, teachers, etc.), including, in particular, users wishing to ground verification tools. This undertaking, however, is purely semantic and does not cover the development of proof rules for Ada or any fragment of Ada.

We discuss their work, below. It is based on combining VDM for sequential semantics with a semantics called SMoLCS (Structured Monitored Linear Concurrent Systems). Our description of the methods, their current shortcomings, and the needs for further research is principally based on self-criticisms published in working papers from the formal-definition project.

4.1.1. Dansk Datamatik and VDM

VDM originated at the IBM Vienna Laboratory in the early 1970's as a method for the systematic development of compilers on an industrial scale. It is a means of defining programming languages, and of specifing and developing compilers, originally targeted for PL/I. During the late 1970's and early 1980's, VDM evolved into a general software development method and has been used in a variety of European countries to develop compilers in addition to database systems, office automation systems, parts of operating systems and language definitions. VDM is probably the most widely used formal method in European industry, although the actual inroads made by such methods are very limited. c[A Our discussion will center around the use of VDM to define the semantics of a programing language (in this case, Ada). The VDM specification language, called Meta-IV, is an enriched form of the lambda calculus. It contains both applicative and imperative constructs and its semantics are informally understood as being "denotational". A programming language is defined by contructing a model of the language in Meta-IV: programming language constructs get their meaning by translation into the model. A proof system does not come automatically, but may be checked for soundness against the semantics.

The VDM method of software development is taught at a number of universities in Denmark, the U.K., Germany, Italy and Poland. Courses aimed at industry have also been initiated. One of the strengths of VDM is that it has in fact been applied to practical projects, including the design of an Ada compiler (in Ada). The compiler project is discussed in [CLEMMENSEN 83], and a general discussion of VDM as an industrial tool is given in [BJORNER 83]. In addition, VDM has been used to define Algol 60, various versions of Pascal, and CHILL (simple CSP-like concurrency features had to be added) and to develop applictions programs such as data base management systems, a parser generator, etc.

A Meta-IV specification looks a great deal like a denotational definition in the style of Scott and Strachey (see [STOY 77]) with many defined combinators added for convenience. Meta-IV has applicative (purely functional) constructs as well as imperative ones (blocks, variables, scopes, assignment statements, loops). It is a purely sequential language. After some experimentation with adding concurrency by combining existing concurrency formalisms (including CSP, SOS, Temporal Logic) the current plan is to use SMoLCS as the vehicle for grafting concurrency onto VDM. Attempts at combining VDM with CSP, SOS, and Temporal Logic are described in [ESPRIT 83, DDC subprogramme]. A plan for using SMoLCS is outlined in [Astesiano 85a].

VDM also has weak points:

1. No formal semantics has been provided for Meta-IV. Not even the static semantics has been fully defined. It would seem reasonable to look for a foundation in the Scott-Strachey style. The most sustained attempt along these lines is [STOY 82] which handles a restricted subset of Meta-IV. This attempt exposed certain difficulties. In particular, the intuition behind informal VDM semantics is that the denotations of program objects are essentially sets, which the objects in Scott-Strachey domains emphatically are not. Bjorner suggests that this difference is rather deep and may pose real problems.

2. Meta-IV has little in the way of nondeterministic constructs. Some features of Ada are nondeterministic, e.g., orders of evaluation, choice at a select statement, "fair" scheduling, etc. Even supposedly deterministic constructs can be overspecified by the construction of a single, purely deterministic, model. For example, the obvious way to model the allocator new for access types is to make intermediate use of a function "find" which returns the value of a new storage location suitable for the object being created by the allocation. Any deterministic implementation of "find" is too specific; for example, either it will or will not reuse old locations, and either answer by itself is the wrong answer. This is a difficulty for all "model-building" (as opposed to, e.g., algebraic) approaches to semantics.

3. The language does not support abstract data types.

4. It does not permit type polymorphism (which is the obvious way to attempt semantics for generics).

5. The language is flat. There is no way to structure specifications hierarchically.

6. There exists little in the way of automated tools. The fact that VDM has been a pencil and paper language has cut both ways. One reason VDM has in fact been used for realistic applications is that the nuisance of proceeding purely formally is removed. On the

other hand without automated support the checking of specifications against implementations can never be carried out in complete strictness.

Note: Difficulties 4, 5, and 6 have been addressed by STC IDEC, Ltd. of the UK, on a grant from the Alvey Commission.

A "classical" paradigm is illustrated in the following paragraphs for the use of VDM to define a programming language.

Meta-IV is first used to formalize the definition of "well-formed program" (essentially, as an abstract data type). This definition can then be used:

- As the specification of a compiler's front end so that, for example, the representation of the context-sensitive syntax as an attribute grammar could be proven correct against the definition of the language, and

- As a precise mathematical description of the input of the denotational "meaning map' (whose target is the set of denotations).

The sequential semantics of individual processes would be defined in Meta-IV and used to specify code generation. The parallel semantics of multiple processes, expressed in terms of some extension of Meta-IV would be used to specify scheduling, synchronization, and run-time support.

Example:

The definition of a "syntactic" domain, i.e., a Meta-IV specification of the syntax of a language. Here we take a trivial language to describe the entities of and operations on a file system.

The domains are:

```
    Command = Insert | Alias

    Insert :: Name File

    Alias  :: Name Name
```

The equation says that the set of commands is the union of the sets of Insert commands and Alias commands. The lines with "::" in effect give the signature of constructor functions. For example, the constructor for "Insert", called in Meta-IV "mk-Insert", is a map Name x File –> Insert. Given a Name n and a File f, the output mk-Insert(n,f) is the command to insert the designated File into the file system under the given Name. The function "mk-Alias" similarly constructs Alias commands.

In this example some terms constructed from mk-Alias are not well-formed commands: a name should not be used as an alias for itself. Accordingly, the boolean function Wf which returns for any Command c the truth value of "c is a well-formed command" is defined as follows

```
Wf(c) = (definition)
     cases c:
        (mk-Insert(.,.)    => true
         mk-Alias(n1,n2)   => n1 /= n2)

type: Command --> BOOL
```

We now turn from a definition of the command language to a description of the file system itself.

The semantic objects of the file system are catalogues (thought of as finite functions mapping names to file ID's) and disks (thought of as finite functions mapping file ID's to files).

```
Filesystem    ::  Cat Disk

Cat                =   Name -m-> Fid

Disk               =   Fid  -m-> File

Name, Fid, File = TOKEN

Operation          =  Filesystem -> (Filesystem[ERROR])
```

Notes: X -m-> Y is the set of finite maps from X to Y. The inclusion of "ERROR" means that the result of an Operation is a pair, whose first element is a Filesystem and whose second element is a signal indicating whether an error occurred, and if so, which. The object nil is the signal that no error has taken place, and in this example, there is only one error condition, the object ERROR.

TOKENS are atomic (i.e., unstructured) objects. The last equation says that the allowed operations on a Filesystem result in file systems (but may, in addition, raise errors).

A file system must be constrained so that all files named in the catalogue reside on the disk, and all files resident on the disk are named in the catalogue. Therefore the definition of the domains includes an invariant which says so.

```
inv-Filesystem(mk-Filesystem(cat,disk)) = (definition)
       rng cat = dom disk
```

Note: here, "rng" and "dom" are the set-theoretical operations of range and domain.

Each command invokes an operation, so one needs a function Elab-Command which elaborates commands – that is, maps Command to Operation. Since the type

Command -> Operation

is the same as the type

Command -> (Filesystem -> (Filesystem[ERROR])).

Accordingly, one can define Elab-Command by defining Elab-Command(c)(fs) for any command c and any file system fs. The definition is by cases on c. We include the case in which c is an Insert command – that is, we assume

c = mk-Insert(n,f)

fs = mk-Insert(cat,disk)

and paraphrase the Meta-IV notation into something a little more like English:

```
if n is an element of dom cat
     then return (fs, ERROR)
     else return (mk-Filesystem(cat',disk'), nil)
         where cat' and disk' are obtained as follows:
             choosing some fid in Fid but not in dom disk,
             set cat'  = cat union (n,fid) and
                 disk' = disk union (fid, f)
```

Notice that the choice function is non-deterministic.

Notice also that this is definition is purely applicative. Meta-IV contains imperative constructs, which could be used, when convenient, to define Ada's imperative constructs. Accordingly, an Ada program written to meet this specification would first be translated into its VDM meaning (via the formal language definition), and one could attempt to prove that that meaning satisfied the specification. To our knowledge the imperative and applicative constructs have not yet been integrated. A technique used to achieve this in other languages has been the reduction of the whole language to some applicative "kernel". The language CIP-L is designed in this way [CIP 85].

4.1.2. The "Genoa/Passau" Group and SMoLCS

As part of the EEC project a group from the Universities of Genoa, Pisa, and Passau is attempting to apply algebraic methods to a subset of Ada which includes the whole of Ada tasking. Giving a technically informative account of SMoLCS is out of the question here. We will summarize certain special difficulties of Ada reported in the two papers [ASTESIANO 85] and give a very general account of their strategy for modeling the whole of the language.

4.1.2.1. Some Difficulties Peculiar to Ada

In Ada, sequential-seeming syntax may disguise underlying parallelism. For example, declarations, expression evaluations, and assignments all may involve actions which are allowed to take place concurrently (or, if sequentially, then in no predetermined order).

Attempts at finding a hierarchical structure for Ada processes run into difficulties because two natural hierarchies within the language—that of task dependence and that of scope—are not always consistent with one another. For example, the distribution of information necessary to handle task termination follows the dependence-structure neatly, but cuts across divisions of scope. A task waiting on an open terminate alternative needs to know the state of its master, but its master is in general invisible.

One ordinarily thinks of the "environment" as something altered only by declarations. On a reasonable understanding of the term, the "environment" in an Ada program can be changed outside declarative regions. For example, an unconstrained variant record has, in effect, a "local constraint" determined by the current value of its discriminant. That constraint can be changed by complete assignment to the record (i.e., outside a declarative part). Another example: let the specification and body of a task type T be textually separated by the text of a package P, whose elaboration activates a task X which will, in turn, eventually activate an instance of T. Execution of X may therefore occur concurrently with the elaboration of program units which textually follow P. A race will then develop: will the body of T be elaborated before execution of X attempts to create an instance of T? If we require that the "environment" of P contain information on the state of the elaboration of T— reasonable, since elaboration of P will eventually call for that of the body of T—then the environment of P is not fully given at the start of the elaboration of P, and can change during the elaboration of P.

The problem of deciding what constitutes an atomic act is non-trivial, especially in the presence of abort statements.

Incorporating the CALENDAR package, which allows explicit reference to time, is non-trivial (and the LRM itself says nothing about the semantics of time). The possibility of such explicit references will ꞏequire refinement of the SMoLCS model of "free-parallel monitoring" for truly parallel execution.

4.1.2.2. The Strategy for Using SMoLCS

Concurrent and sequential features in Ada are mixed in complicated ways. The strategy for applying SMoLCS analyzes Ada texts in two steps. The first can be thought of as "preprocessing" the text into a target language which makes explicit the concurrency hidden in seemingly sequential operations. For example, the non-deterministic possibilities involved in evaluating a shared variable (i.e., one shared between tasks) are captured by translating the evaluation into what Milner calls an indexed sum, a kind of disjoint "sum" of the possible values the variable might receive.

The second step assigns a semantics to the new language, and can itself be subdivided: an operational, algebraic specification of the "transition systems" defined by the new language; and an observational semantics equating two algebraic terms (which correspond to programs, tasks, etc.) if they exhibit the same "observational behavior" in every program context.

The preprocessing can be described in a style that looks like definitions in ordinary denotational semantics (and in fact can be given an alternative interpretation as a denotational evaluation, rather than as a translation).

Concurrency is interpreted algebraically, in a formalism something like that of Plotkin's SOS. [PLOTKIN 82] formulas (the simplest kind) look like:

condition => s -f-> s'.

This means that if the Boolean expression 'condition' is true, then the (flagged) transition from s to s' belongs to (is an allowed transition of) the system. The flag 'f' contains information about the transition, such as synchronization information.

The behavior of the system is specified as the result of deductions from rules. Equivalently, among all models of the system a canonical "minimal" model is singled out in which the denotations of distinct terms are equal if the terms are provably equal on the basis of the rules. Here is an approximate example of a rule: if the transition b1 to b1' can occur, flagged by "process1 sends value v to process2" and the transition b2 to b2' can occur, flagged by "process2 receives value v from process1", then the parallel states b1|b2 can make a transition to the parallel states b1'|b2'.

The models of execution can be factorized to "concurrent algebras" by introducing a notion of observational equivalence, identifying those elements which can't be distinguished observationally. Various choices of what is and what is not to count as "observable" are possible.

We should note that the models can be parameterized by the kinds of system-specific information to which Ada programs may refer, such as duration, storage size, etc.

4.2. Acceptance Standards for Verification Environments

4.3. Standards for Accepting Verification Systems

4.3.1. Acceptability

What does it mean to stamp a program, or program-specification pair VERIFIED? Certainty is unobtainable and verifications will always be subject to certain assumptions. The best one can do is to reduce the strength of those assumptions and to make them explicit. That is what a certification scheme could achieve. One could even imagine grades of certification, like the classification scheme for multi-level secure systems (a possibility we won't pursue here).

It will be useful to separate verification systems schematically, into two parts:

* The "verifier": A mechanism for granting the stamp VERIFIED to suitable input pairs of the form (program, specification), possibly making use of externally supplied information.

* The "manager": Responsible for storage and retrieval of fully or partly verified programs, packages of mathematical lemmas, etc., and also for performing certain useful but logically inconsequential modifications upon them.

In general, the manager manipulates data objects which are produced (interactively) by the verifier. The manager must maintain the "logical" integrity and security of the data objects. For example,

- Nothing except the official stamping mechanism should be able to stamp a program VERIFIED.

- Any alteration of a verified (program, specification) pair should cause the pair to lose its VERIFIED stamp, and if the specification is altered, then all programs logicially dependent on that specification must lose their stamps.

  Exception: Certain trusted processes, meaning-preserving transformations, would be allowed to modify data without affecting its stamp. For example,

  - Reformatting for pretty-printing

  - Systematically changing identifiers

  - Stripping the comments from a program (or, a much more sophisticated operation, removing the debugging flags)

  - Generating and appending an English-language "translation" to the formal specicifications

  - When a verified program is retrieved, we must be sure that the code retrieved is actually the code which has been verified (no substitutions).

A serious standard for accepting verification systems will require certain preliminary work:

- Ada syntax must be officially "mathematized" and the notion of "legal program" given an official formal definition via an attribute grammar, or as an abstract data type, or whatever. In principle this is a routine undertaking, but a standard would have to be settled on.

- The meaning of Ada (or, at least, of some fragment of Ada for which one hopes to produce verification systems) must be established by an official formal semantic definition.

- A standard specification language for Ada (with a formal semantics) must be settled on; its assertions are what the verifications will be about. The specification language should be rich enough to specify the property of "being a verification system."

Part of the agreement on semantics for both programming language and specification language is agreeing on what it means for a program to "meet" a specification, in particular, whether the notion of "meeting a specification" will be split into various related sub-notions.

Consider: In much of the literature on program verification, the specification language is a first-order language for making assertions about instantaneous "states" of the system, and programs are specified by their input-output behavior: if the state satisfies condition A before execution, then it will satisfy condition B after. (Strictly speaking, the specifications aren't individual Boolean expressions like A and B, but pairs of such expressions.)

Two possible notions of meeting the specification (A,B) immediately arise: "weak correctness" which assumes that the program will terminate (and says nothing at all if the program does not); and "total correctness" which asserts termination. These notions are compatible, in the sense that "meets (total)" implies "meets (weak)", and a system which operates on partially correct specifications can "downgrade" specifications which are total and then use them. It has sometimes been proposed to distinguish normal executions—those in which exceptions (or, perhaps, predefined exceptions) are raised and those in which they are not. A specification like "(A,B) totally for all normal executions" is logically incomparable to "(A,B) weak."

If we wish to make assertions about something other than a pair of instantaneous states the possibilities multiply. All one can say abstractly is that input is supplied to a verifier in the form "The following semantic information is allegedly true" and the verifier is given the opportunity to say "Agreed." The logically necessary minimum for accepting a verifier is suitable evidence that:

> (*) whenever the proposition "program P meets specification S" is verified by the system, it is indeed true according to the official semantics that P meets S.

In particular, the system's power or convenience are irrelevant, just as the efficiency of its generated code is irrelevant to the validation of a compiler. A system which never stamps anything VERIFIED automatically satisfies (*).

Condition (*) does not say anything about the internals of the verifier, or demand that its underlying language be identical with the official specification language. It is merely an input-output assertion about the verifier—that whatever "official" specification the verifier certifies is true.

4.3.2. An Example

In the next section, we consider what kind of evidence of the correctness of a verification system might be acceptable. To make the discussion more concrete first we offer, as a paradigm, an imaginary example.

Cast of characters:

Customer -- who wants an applications program meeting, and verified to meet, specification S.

Ver -- a "certified" verification system, written in Ada.

Supplier -- who undertakes to provide P, a program which verifiably meets specification S.

We imagine that Ver, written in Ada, is widely distributed and highly portable. Supplier may therefore obtain a copy of Ver, and use it to do his verification of P. Even omitting outright chicanery, or errors in the Ver source code (supposedly ruled out by the certification of Ver), there are many reasons why Supplier's verification could be mistaken: an error in Supplier's compiler or run-time support, a failure of security in the operating system, etc. Accordingly, Customer should be able to re-do the Supplier's verification by running Ver on his own system.

This means two things. One: in the ideal case, there will exist machines for which verified compilers have been written, the customer will own such a machine and such a compiler, and Ver will have been written in a subset of Ada which Customer's compiler is verified to handle correctly. Actually, we will need less: we need only require that the "core" of Ver (see below) be written in a subset of Ada for which the compiler is verifiably correct.

And two: the supplier is not actually doing the verification himself, even though it feels to him as though he is. He's really preparing a big file containing all the evidence needed by the? cUSTOMER's version of Ver to verify P in one batch run (in effect, a machine-readable form of the proof). We will call this file the "support" of the verification, a neutral term which avoids prejudging the nature of the support.

This picture of the supplier's activity makes it clear that the only part of Ver that needs to be correct is a "core" verifier that provides no user support at all: the core needn't contain the tens of thousands of lines necessary to provide window packages, for example, or to generate "proof tacticals." Mistakes in these (practically essential but logically irrelevant) features can at worst result in inadequate support that will be rejected by the customer's "core".

The input to the customer's core is the triple (program, specification, support). The support is specialized information, meaningful only to some particular verification system. It is reasonable to require of the core

(*-core) the triple (program, specification, support) will be accepted by the core if and only if (understood in the light of a logically sound system) the support does indeed guarantee that the program meets its specification.

The logically necessary minimum is obtained by replacing "if and only if" by "only if".

Note: When Ver is running on the customer's system, the core of Ver must be isolated using security-style techniques to prevent circumvention. Among the things a Trojan horse could do is issue the VERIFIED stamp itself and trick the core into issuing VERIFIED by showing it different input.

The supplier could partly test the results at his end, by running this triple through his own core. If the supplier's system supports an isolation of the core from external sources of error, so much the better.

Precisely what would the supplier send to the customer? (What would the support look like?) Consider a familiar case: Ver does the verification in the style of the following embedded assertions.

- The most primitive possibility is that the supplier sends a fully annotated program, with no proof steps omitted, and, to make life easier, comments indicating which rule was used in each step. The support would be nothing but a complete derivation in Hoare-logic, and the customer's core need only be a set-by-step proof-checker. Unfortunately, the sheer size of the "deliverable" might become unmanageable.

- A more sophisticated possibility is this: whenever the supplier's system has invoked an automatic technique to fill in part of the proof (VC generation itself is such a technique), the support simply includes an instruction to the Customer's core to invoke the same technique. The cost of this is of course the need to build and certify a larger core.

- The documentation produced by the previous methods could be improved by pruning the proof tree of useless steps, either by a human user or automatically.

- Errors committed by the supplier's system may cause the {supplier's Ver to validate a program incorrectly, or to generate incorrect support for a legitimate verification (causing the customer's core to reject it), but cannot cause the customer's core (assuming it is correct) to certify a program improperly. If it happened that the customer's core corrected the mistakes of the supplier's Ver, that would cause no problem. Example of a correction: the supplier's Ver incorrectly implements an automatic simplification technique and gets the right simplification by the wrong means; the same technique, correctly implemented, is invoked in the customer's Ver by the support and arrives at the right simplification by legitimate means.

4.3.3. Evidence

As noted above, both (*) and "(*-core) + suitable security" are minimally sufficient specification of a verifier. What kind of evidence of their truth would be acceptable? Even leaving aside the oddity of accepting a verifier on the basis of case-by-case testing, it's hard to see what a meaningful sample of test-cases could possibly be.

The notion of "suitable evidence" should not be defined so as to prejudice the kinds of systems that could be built. That's the reason for calling the documentatior produced by the verification system a "support" rather than a "proof".

In general we expect the proof of (*) to consist of two steps: the ordinary mathematical proof that a certain algorithm, if implemented, will guarantee (*) and the implementation of that algorithm. Judgment will be required to draw the line between those arguments which are appropriately presented by hand (roughly, things with a high conceptual content) and those most appropriately checked by machine (things with lots of detail), and such judgment would have to be exercised anew for any proposed verification system.

If, for example, the system proceeds by generating and checking verification conditions based on Floyd-Hoare style rules, then part of the proof of (*) will be a proof of the soundness of those rules. Those soundness proofs, based on direct appeals to the underlying semantics, will be part of the demonstration that certain checking algorithms will not accept fallacious inferences. We expect these soundness proofs to be pieces of ordinary mathematics.

Consider now the problem of implementation: how could we verify the source code of Ver's core or verify the customer's compiler? It's conceivable that a very primitive core could be meaningfully verified by hand (and perhaps cross-checked on uncertified systems), and that a compiler sufficient to compile the core could be generated directly from the formal definition of some fragment of Ada (if the definition were in the proper style). The compiler's efficiency would not be an issue. The rest of Ver could be piled high with the most powerful user support available and understood as mechanical help in preparing support to be checked by the core.

In a better-than-average of all possible worlds, it might then be possible to push enough verified code through this primitive core to bootstrap some extra power into the core itself, or into the compiler, or into the security apparatus.

Security for the core need only be primitive, since one could run the core on a dedicated machine, under a very simple operating system. In one sense, security for the manager would be simple to design: an access table would list which manager programs were allowed to modify verified programs without causing their VERIFIED stamps to be automatically revoked; and a graph would keep track of logical dependences among verified programs. However, specifying and verifying the trusted processes—for example, one which removed all the debugging flags and left a program semantically equivalent to one in which all flags had been set to false—could be much more difficult.

4.4. Bibliography

[ASTESIANO 85A] Astesiano, E. and Mazzanti, F., Reggio, G.
"Towards a SMoLCS based abstract operational model
for Ada," 1985.

[ASTESIANO 85A] Astesiano, E. and Reggio, G. "A Syntax-Directed
Approach to the Semantics of Concurrent Languages,"
Istituto di Matematica, Universita di Genova,
Italy, 1985.

[BJORNER 82] Bjorner, D. and Jones, C., (eds.), Formal
Specification and Software Development,
Prentice Hall, 1982.

Bjorner, D. and Oest, O., (eds.), Towards a
Formal Description of Ada, LCNS 98,
Springer Verlag, 1980.

[BJORNER 83] Bjorner, D. and Prehn, S., "Software Engineering
Aspects of VDM," in Theory and Practice of Software
Technology, Ferrari, Bolognani, and Goguen (eds.),
North-Holland, 1983.

[CIP 85] CIP Language Group, The Munich Project CIP, Vol. 1,
Lecture Notes in Computer Science 183,
Springer Verlag, 1985.

[CLEMMENSON 83]Clemmenson, G. and Oest, O., "Formal specification
and Development of an Ada Compiler – a VDM Case
Study," Dansk Datamatik Center, 1983.

[DONZEAU 80] Donzeau-Gouge, V., Kahn, G., Lang, B., Formal
Definition of the Ada Programming Language, INRIA,
1980.

[ESPRIT 83] ESPRIT Preparatory Study, "A Critical Appraisal of Formal
Software Development, Theories, Methods, and Tools"

(Short title: 2 vols.: STL subprogramme, Standard Telephone Laboratories, UK; DDC subprogramme, Dansk Datamatick Center, Denmark. Short Title: Formal Methods Appraisal.

[PLOTKIN 82] Plotkin, G., "An Operational Semantics for CSP," in IFIP TC2 Working Conference on Formal Description of Programming Concepts II, Garmisch, 1982.

[STOY 77] Stoy, J., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977.

Stoy, J. "Mathematical Foundations," in Formal Specification and Software Development, Bjorner and Jones (eds.) Prentice-Hall, 1982.

5. SURVEY OF NEEDS FOR FORMAL VERIFICATION

5.1. Development of the Questionnaires

Our approach to determining user needs has been to develop and circulate a questionnaire among people with experience in programming large systems. This questionnaire is meant to find out two things:

- What sort of system functionality is the most critical? That is, for which functions would formal verification most increase reliability?
- Which Ada language constructs are the most heavily used?

Formal program verification is basically unknown and barely used in the software development process. We did not simply ask, "do you have a need for formal program verification?" because that would probably have evoked no response at all. (In fact, in military software development, the phrase "formal verification" sometimes refers to the act of the military customer certifying that a battery of tests has been run on the software.) Instead, we tried to find those aspects of the software product the developer is willing to spend the greatest amount of money to test. This may be the surest indication of a candidate for formal verification.

Correct rules of proof are not known for all of Ada, but the scope of Ada verifiability can certainly be increased by research. Thus, we also set out to find out directions in which future research would be most useful, i.e., which Ada language features not now known to be verifiable do software engineers feel are most critical?

Initially, a draft form of the questionnaire was prepared. This draft was sent to a number of people for comments. We chose roughly a dozen individuals whose opinion we respect, some from the verification community, some designers of large software systems. After collecting comments from these people, we modified the questionnaire, incorporating many suggested improvements. The modifications roughly doubled the length of the questionnaire. This first questionnaire was distributed and then reported on at the First IDA Workshop on Formal Specification and Verification of Ada, in March, 1985. The questionnaire was subsequently expanded somewhat and this second questionnaire was distributed widely.

The questionnaire divides into two parts:

- General systems development experience
- Ada-specific experience

The first part asks for information about a system the respondent has worked on, not necessarily involving Ada. The questions pin down the type of system developed, its size, languages and tools used, and a brief statement of its purpose. The questions then try to determine how much testing effort was or is expected to be devoted to the project, and in what specific areas the greatest fraction of effort was devoted. The point is, if a developer is going to spend dollars on verification, what critical functions, modules, or features will be deemed the most important to verify?

The second part asks questions about the use of Ada: which language constructs are currently used, which are never expected to be used, and which are avoided now because of lack of faith in the particular compiler used.

The questionnaire is not limited to a survey of Ada users; nevertheless, we decided to concentrate on the Ada community for the first mailing. Our main source of contacts was a list of current Ada contracts compiled by Ann Reedy and published in The Ada Letters. We telephoned most of the organizations on that list, both to determine the most appropriate recipient of the questionnaire, and to ask knowlegable people in the Ada community for other potential contacts. The second questionnaire was sent to 120 people on the IDA Ada verification list, and distributed at the SIGAda November 1985 meeting in Boston. Roughly 650 people at the meeting took a copy of the questionnaire.

5.2. Results of the Surveys

The first questionnaire was sent to key people in the following organizations. Eighteen individuals in 15 organizations responded. The last of these questionnaires was sent out on Feb 22, 1985.

Organization	Project	# responses
Singer/Librascope	front-end for TACFIRE	1
Singer/Librascope	message communication terminal	1
Dalmo Victor Operations	tank sensor integration	2
Veda	generic message editing	2
Tasc	ASAP	
McDonnel Douglas	CAMP	
McDonnel Douglas	porting ICSC Ada	
McDonnel Douglas	convert AIS to Ada	1

Ford Aerospace	G-3 Maneuver Control	1
General Dynamics	TAG	
General Dynamics	decision support system	1
General Dynamics	IMF	
GTE	WIS	
Magnavox	AFATDS	1
Harris Corp.	ALPC	
Sonicraft	MEECM	
LTS	MEECM	
NAV AIR	F-18 operational flight program	1
NAV AIR	aircraft control & HUD	1
Syscon	ACCAT GUARD Ada reimplementation	1
RCA	MCF RTM O/S / ASOS	1
System Development Corp.	STARS	2
TRW	STARS	
TRW	prototype advanced APSE	
TRW	ASOS	
TRW	TDBMS	
Computer Corp of America	Ada DBMS	
Intermetrics	hardware description lang. analyzer	1
Intermetrics	S/370 Ada compiler	1
Telesoft	WIS compiler	
SofTech	Ada/M UYK-44 retarget	
NYU	Ada/Ed	
Florida SU	Cyber 170 Ada compiler	
UC Irvine	Arcturus	

The second questionnaire, despite being distributed to over 770 people, elicited only 8 responses. Clearly the personal contact involved in the previous distribution played a large part in motivating people to respond. Due to several anonymous replies (an option incorporated as a result of suggestions at the march, 1985 workshop), the following list of organizations represented in the responses is somewhat limited.

Organization	Project
*	DOS for 1750A
*	interface laser printer to IBM channel
ACT	Ada-to-MILSTD 1750A cross compiler
*	DRDB

Organization	Project
*	simulation of fire control system/ command management system
STC	call control for switching system
PRIOR Data Sciences	re-usable Ada components
*	FORTRAN & COBOL to Ada translator

Several interesting results have emerged:

* Many Ada systems have interfaces to other languages. The fore:gn languages were various assembly languages, FORTRAN, and in one case, PASCAL.

* Correctness and precision of floating-point computations are not large problems for testing.

* Denial-of-service problems receive relatively less testing effort than timing constraints.

* One respondent suggested that a concern about erroneous programs was "academic nit-picking." However, several others had encountered erroneous programs or programs with incorrect order dependences. (In one instance, this was documented and left in the final code.) It's possible that the true rarity is not writing an erroneous program, but realizing that an erroneous program is erroneous.

* Absolutely no respondent uses or claims to have an urge to use tasks passed as parameters to subprograms.

* Few Ada users can make do without access types.

* Many Ada users can make do without using functions with side effects.

* Two-thirds of the respondents make use of recursive subprogram calls.

A copy of the questionnaire, with total numbers of responses filled in, follows. Rather than include both questionnaires separately, due to the small number of responses to the second, the results have been combined. Two sections of the questionnaires were sufficiently different that we were not able to combine the results conveniently. Instead, both versions are presented, appropriately labelled.

5.3. The Questionnaires

Section I. (Both Questionnaires. Total responses: 26)

Please answer the questions below with reference to a specific software development project that you are or have been engaged in. If you cannot answer from experience about a project involving Ada, we are still interested in any experience with a medium-scale to large-scale software project.

a) Roughly, what is the size of the project, in bytes ?	40 KB - 60 MB
b) To which hardware is it targetted ?	
c) In what language(s) is it written ? What fraction for each ? (or give rough numbers for lines of code)	Ada:21 FORTRAN: 2 Pascal: 1 assembly lang.: 10 other:7
d) Was a program development language (PDL) used ?	NO: 7 YES: 10 Ada: 5
e) Is the project a commercial product development, DoD contract, IR&D, or other ?	DoD: 14 IR&D: 8
f) Describe briefly the goal of the project.	

Section II. (Both questionnaires. Total responses: 26)

We are interested in estimating the potential needs for formal verification in such a project. Because formal verification is not now a common phase of software development, we would like to gauge the most likely applications for formal verification by finding the areas to which the greatest fraction of testing now goes. For each area below, if it relates to the project you described above, please indicate the relative fraction of the testing effort devoted. Feel free to add any other areas which consume significant testing resources.

Level of testing effort:	none	very little	some	very much
a) timing constraints -- verification that real-time limits are not exceeded due to computational complexity	[7]	[7]	[6]	[6]
b) space limitations -- verification that space bounds are not exceeded due to dynamic memory allocation, or stack overflow as a result of nested procedure calls or interrupt handling, etc.	[3]	[8]	[6]	[8]
c) protection of sensitive data from unauthorized disclosure	[9]	[7]	[5]	[5]
d) protection of data integrity	[5]	[4]	[9]	[7]
e) resource management	[6]	[8]	[9]	[3]
f) denial of service	[7]	[13]	[3]	[2]
g) real-time external device control with feedback	[13]	[2]	[5]	[6]
h) fault tolerance	[7]	[5]	[8]	[5]
i) floating-point numerical computations: correctness and precision	[15]	[5]	[5]	[0]
j) fixed-point or integer numerical computations	[8]	[6]	[8]	[3]
k) machine-dependent interfaces, perhaps using low-level Ada	[7]	[5]	[5]	[9]
l) parallel processing (concurrency; tasking)	[8]	[5]	[4]	[9]
m) handling of external interrupts	[6]	[3]	[11]	[6]
n) graceful recovery from errors in external input	[1]	[5]	[11]	[8]
o) graceful recovery from internal program errors -- logical design problems, hardware	[2]	[5]	[11]	[6]

failures, etc.

Level of testing effort:	none	very little	some	very much
p) independent module testing	[0]	[2]	[15]	[9]
q) integration of system modules, each independently reliable	[0]	[2]	[6]	[17]
r) operations involving complicated (e.g. nested) data types	[3]	[6]	[7]	[9]
s) portability	[8]	[7]	[4]	[7]

t) other -- please explain

- inter-process communication in a shared bus architecture
- mutual exclusion of processes using shared resources (race conditions and deadlocks)
- generics - validation of the "correctness" of a generic definition

Section III. (First questionnaire. Total responses: 18)

The following questions are Ada specific. We realize that there are now compilers in use which implement only a portion of Ada, or which may not implement esoteric language features in an efficient manner. Which of these Ada language constructs do you find are heavily used, avoided because your compiler is not adequate, or not used at all ?

	heavily used now	don't trust compiler	not used & not likely to be used	
a) low-level Ada:				(*)
address clauses	[6]	[2]	[4]	2
unchecked storage deallocation	[4]	[3]	[4]	2
unchecked type conversion	[5]	[3]	[4]	1
b) interfaces to other languages	[7]	[1]	[6]	1
c) generics	[3]	[6]	[4]	2
d) recursive constructs:				
types	[9]	[0]	[5]	
subprogram calls	[10]	[0]	[5]	

e) exception handling	[11]	[3]	[0]	
f) tasking,	[3]	[4]	[4]	
including, in particular:				
shared variables	[2]	[4]	[8]	
tasks passed as parameters to subprograms	[0]	[6]	[8]	
task and entry attributes	[2]	[5]	[7]	
dynamic task creation	[2]	[3]	[9]	
g) functions with side effects	[2]	[0]	[13]	
h) global variables, except in packages	[9]	[0]	[5]	
i) limited private types	[5]	[1]	[7]	1
j) subtypes of predefined integer types	[11]	[0]	[2]	
k) subtypes of predefined real types	[7]	[1]	[6]	
l) access types	[10]	[1]	[1]	
m) renaming declarations	[6]	[1]	[6]	

(*) is "not available"

Section III. (Second questionnaire. Total responses: 8)

The following questions are Ada specific. Indicate the amount of use of the Ada language concepts below. For the constructs you have used, please indicate whether you could avoid using them without a significant increase in programming hours. This may be difficult to answer; leave a blank if you are not sure. For the constructs you have not used, give the reason from the following list:

1. the construct is not implemented by the compiler

2. the construct is not correctly implemented by the compiler

3. the construct is not efficient

4. the construct is unfamiliar to the programmers

5. the construct is not needed for this particular project

Section IV. (Both questionnaires)

In your experience, how commonly used are the following SLPPRESS pragmas ? Why were they used (if they were) ?

	never	rarely	some	often	why?
ACCESS_CHECK	[14]	[1]	[1]	[3]	
DISCRIMINANT_CHECK	[14]	[0]	[2]	[3]	
INDEX_CHECK	[13]	[0]	[2]	[4]	
LENGTH_CHECK	[13]	[0]	[2]	[4]	
RANGE_CHECK	[13]	[0]	[2]	[4]	
DIVISION_CHECK	[14]	[1]	[1]	[3]	
OVERFLOW_CHECK	[14]	[1]	[1]	[3]	
ELABORATION_CHECK	[15]	[0]	[1]	[3]	
STORAGE_CHECK	[14]	[1]	[1]	[3]	
			answer:	to increase execution speed	

Section V. (Both questionnaires. Total responses: 26)

The Ada Language Reference Manual defines certain compiler-dependent situations in the following way:

- Erroneous program: Compilers are not required to detect violations of certain semantic rules of Ada, either at run-time or compile-time. For example, the results of procedure calls should not depend on the method of parameter passing, as it might if parameters are aliased. Programs which violate these rules are called erroneous.

- Incorrect order dependence: A rule of the Ada language under which different parts of a given construct are to be executed in some order that is not defined by the language (but not executed in parallel), and execution of these parts in a different order would have a different effect. The compiler is not required to provide either a compile-time or run-time detection of a violation. An example would be evaluation of the expression "f(x) + g(y)", where, due to side-effects, the sum would depend on the order in which f and g are evaluated.

				(if used)
	not used	used	used	could
	(give reason #)	some	heavily	avoid
				easily
a) low-level Ada:				
length clause	[1. 5.x2 other]	[1]	[1]	[N]
enumeration/record representation	[5.x3]	[1]	[2]	[N]
address clauses	[1. 5.x3]	[1]	[1]	[N]
unchecked storage deallocation	[5.x2]	[2]	[2]	[N Y]
unchecked type conversion	[5.x2]	[2]	[2]	[Nx2 Y]
b) interfaces to other languages	[1. 5.]	[4]	[0]	[Nx2]
c) generics	[5. 2. 1.]	[0]	[4]	[ ]
d) recursive type definitions	[5.]	[1]	[3]	[Nx3]
recursive subprogram calls	[5.x2]	[1]	[3]	[Nx2]
e) exception handling	[3.]	[ ]	[5]	[N Y]
f) tasking,	[5.x2]	[2]	[1]	[N Y]
including, in particular:				
shared variables among tasks	[5.x3]	[ ]	[ ]	[ ]
tasks passed as parameters	[5.x3]	[ ]	[ ]	[ ]
task and entry attributes	[ ]	[2]	[1]	[Y]
dynamic task creation using NEW	[5.]	[2]	[ ]	[N]
nested accepts	[5.x2]	[1]	[ ]	[ ]
g) functions with side effects	[5.x2 other]	[1]	[1]	[Y]
h) global variables, except in packages	[5.x2 1.]	[2]	[1]	[N]
i) limited private types	[1.]	[3]	[2]	[Yx2]
(not limited) private types	[1.]	[2]	[3]	[Yx3]
j) subtypes of predefined integer types	[ ]	[5]	[1]	[Yx2]
subtypes of predefined real types	[5.x3]	[3]	[ ]	[Y]
k) access types	[5.]	[2]	[3]	[Nx2]

	never	rarely	some	often
a) How often have you encountered erroneous programs ?	[9]	[5]	[4]	[2]
b) How often have you encountered programs with incorrect order dependence ?	[13]	[5]	[2]	[0]

Second questionnaire (total responses:8)

If you have encountered programs which were either erroneous or had incorrect order dependence, please explain something about the circumstances.

Answer: erroneous program encountered with task-scheduling assuming a "round robin", time slicing approach.

Did you ever make use of special knowledge of a particular compiler to understand the behavior of such compiler-dependent features and retain them in the final code? (elaborate)

Answer: yes, though documented.

Index

APPENDIX A

Adapting the Gypsy Verification System to Ada
John McHugh, Karl Nyberg

Verifying Ada Programs
Raymond J. Hookway

Adapting the Gypsy Verification System to Ada

Workshop on Formal Specification and Verification of Ada
Institute for Defense Analysis
18-20 March 1085

John McHugh - Research Triangle Institute
Karl Nyberg - Verdix Corporation

1. Introduction

DoD directive 5000.31 [DoD] requires that new mission critical computer programs written for the department of defense be written in Ada[1] [Ada]. The statutory definition of mission critical (10 USC 2315) includes security applications specifically. Computer security has been one of the principle driving forces for applied verification work in recent years. These factors lead us to one of two conclusions: 1) The time is rapidly approaching when it will be necessary to apply verification techniques to programs written in Ada; or 2) DoD 5000.31 will have to be modified to exclude secure systems. While there exists a well known antipathy towards Ada within parts of both the verification and the computer security communities, it is unlikely that the DoD policy towards Ada will undergo substantial change in the near future. If this is the case, it will be necessary to develop an Ada verification capability in the near future.

There are several ways in which such a capability could be developed. A first option would be to start from scratch, using any of the formal models of program specification and verification and build a system specifically designed to verify Ada programs. A second option is to ignore the Ada specific aspects of the problem entirely. Under the current certification criteria of the DoDCSC, it is not necessary to deal with the implementation language for a system in a formal manner, so it could be argued that current systems are just as suitable (or unsuitable) for Ada as for any other language. In this case, it is only necessary to provide a convincing argument for the conformance of the Ada implementation code to the verified formal top level specification of the system in question. Finally, it is possible to adapt an existing verification system to deal with Ada.

The first approach is possible, but would take an excessive amount of time and resources. Current verification systems represent investments of ten or more man years each, expended over periods of five to ten years. The second approach is representative of the practice followed for the Honeywell SCOMP, a product currently approaching A1 certification by the DoDCSC. It appears that the requirement for a convincing argument concerning the equivalence of the FTLS and the implementation resulted in an extremely complex and concrete FTLS and greatly increased the verification effort. Being able to verify an Ada based FTLS for an Ada based implementation should

[1] Ada is a registered trademark of the Ada Joint Project Office.

obviate these difficulties. Additionally, there is substantial interest in systems which go beyond the A1 criteria by requiring code verification for which second approach would not be viable. The third approach offers a chance to capture much of the investment in a current verification system while gaining experience with the verification of Ada. We argue for such an approach, based on the Gypsy [Good78] system, suggesting that it will lead to a prototype code verification system for Ada with minimum (although not insubstantial by any means) effort. Taking advantage of the Ada packaging mechanism, we feel that verified packages can function within a larger Ada environment, making possible the implementation of security kernels and the like.

The remainder of the paper discusses some of the problems associated with the verification of Ada, suggests ways in which these problems might be addressed, and indicates ways in which the Gypsy system could be combined with the front end of an Ada compiler and transformed into a prototype system for the verification of Ada.

2. Trouble spots in Ada

Although one of the early design objectives for Ada (in the days when it was still known as DoD-1) was to facilitate proofs of program properties, the committee nature of the requirements process resulted in a language which was required to carry a certain amount of the baggage of 1960s style programming languages. Among the potentially most troublesome of these are the presence of arbitrary control flow constructs i.e. the "go to" statement, and unrestricted access to global variables which, in addition to complicating proofs about sequential programs, render concurrent programs intractable under many circumstances. Other features of the language include the possibility of side effects from function invocations, exceptions during expression evaluation, and the lack of an explicit evaluation order for the operators of an expression. These factors, combined with the lack of a formal definition for the semantics of the language, have lead some workers to despair of verifying any aspect of the language. Indeed, it has been noted that given the proper Ada context, it may be impossible to prove anything about the value of X after the execution of so simple a statement as

```
X := 1;
```

We maintain that the situation is not quite as grim as indicated above. Just because a language contains a particular feature does not mean that all programs written in the language will contain that feature. The adverse interaction among features of the language, does not mean that all of them must be discarded, or that all occurrences of a feature in a given program are intractable. Although the word "subset" is an anathema to the Ada world, we feel that a useful set of Ada constructs and programming practices can be defined in such a way that realistic and functional programs can written and verified using them. Although the task is substantially more difficult, because of the extra complexity of the language, we feel that a theory of verifiable Ada can be developed in much the same way as Boyer and Moore developed their FORTRAN

[Boyer80] theory. Platek [Odyssey84] and his colleagues at Odyssey Associates have recently defined an initial subset of Ada which they feel is suitable for verification. One feature which they rule out is the exception mechanism. We feel that the Ada exception mechanism is sufficiently like the Gypsy mechanism so that its verification is tractable, and we propose to include exceptions in our system.

Ada as currently defined has no specification mechanism. While it is possible to use an external specification mechanism, i.e. one in which the program and specification are joined only during the verification process, we are more comfortable with an internal mechanism, similar to that used in Gypsy. At the same time, we would like our verifiable code to be acceptable to a variety of Ada translators. An extension of Luckham's Anna notation [Luckham84] to accommodate exception returns from routines appears to be the most promising mechanism available at the present time, although a specification language using the Ada PRAGMA construct cannot be ruled out.

3. A hybrid system

We propose to base our prototype Ada verification system on a combination composed of an existing Ada compiler and an existing verification system. The Ada compiler is the one developed and recently validated by the Verdix corporation of McLean, Virginia, while the verification system is the Gypsy Verification Environment, developed at the University of Texas. There are several reasons for the choice of such a hybrid system. Ada is a large language with a complex syntax and semantics. Using an existing front end from a validated compiler eliminates much of the effort required to implement a front end for the verification system. It also provides a direct method for providing executable versions of the verified programs, as well as facilitating systems which contain mixtures of verified and unverified programs. The use of a modified version of the GVE as a back end for the Ada verification system offers similar advantages. We feel that the initial set of Ada constructs which can be verified will be roughly equivalent in power and flavor to the Gypsy language. Previous efforts to model Ada constructs in Gypsy [Akers83], and vice versa provide evidence for this assumption. Although Ada type rules are "stronger" than those for Gypsy, it is possible to write Gypsy as though it were typed like Ada. The Gypsy exception mechanism, though somewhat more tractable than the Ada exception mechanism is suitable for modelling Ada. Most of the Ada operators are already present in Gypsy.

The proposed hybrid consists of three primary components, the Ada front end, the intermediate form translator, and the verification back end. Each of these are described briefly in the sections which follow.

4. The Ada front end

As noted above, the front end of the proposed system is based on the parser and semantic checker of an existing, validated, Ada compiler. The parser and semantic checker will require some modifications to accept Ada with embedded specifications. The output of the modified front end will consist of the compiler's internal representation of Ada programs, extended to include the specification constructs. Assuming that a specification language such as Anna is chosen, these modifications should be relatively straight forward. The internal representation will be captured at a stage in the compilation process where name resolution has been performed and operator overloading has been removed so as to simplify subsequent operations.

5. The intermediate form translator

The intermediate form translato serves a dual purpose. Its primary function is to convert the Ada compiler's representation of a program into a representation which can be entered into the verification back end as though it were the output of the Gypsy parser. Its secondary function is to ensure that the code to be verified conforms to the set of constructs acceptable to the verification system, i.e. that the program to be verified is in fact written in the verifiable Ada subset. Given that both the Ada front end and the Gypsy back end use internal representations which are abstractions of prefix trees, the translation operation is a straightforward, if complex, syntactic one. The enforcement function, on the other hand, may involve substantial semantic analysis. It is hoped to simplify both of these tasks by taking advantage of utilities, already present within the front end, for manipulating the internal form of Ada programs.

6. The modified GVE

The output of the translation process will be a Gypsy-like representation of the Ada code to be verified in a form suitable for loading into the modified GVE. Once such an Ada database has been restored into the GVE, verification conditions can be generated and proved in the same way these steps are performed for Gypsy programs in current versions of the system. To support Ada verification, substantial modifications will be required for a number of components of the GVE. The verification condition generator will require modification to reflect the semantic differences between Ada and Gypsy statements. In a similar fashion, the expression simplifier will also require modification and extension. The prefix to infix conversion routine, used to display internal forms to the user will be modified to use an Ada syntax. We hope to take advantage of the previous work on a Gypsy to Ada translator for much of this step. It is hoped that the prover will require little or no modification. Modifications to the top-level or user interface to the system should be restricted to the removal of unneeded functionality and system components such as the optimizer and code generators.

7. Summary and conclusions

We have proposed a prototype Ada verification system based on a hybrid of an existing compiler and verification system. Although such a system is not capable of supporting verification of the entire Ada language, it is claimed that it will support a language comparable to those now being verified and suitable for similar programs. While the construction of such a system involves a substantial effort, we are confident that the effort is much less than that involved in building a verification system for Ada from scratch. A hybrid system, such as we propose, will allow the verification community and the growing applications community it supports to obtain experience with Ada verification in the near future. Such experience will provide a sound basis for future revisions of the language to support verification should this prove necessary or desirable.

8. References

[Ada] - *Ada Programming Language,* ANSI/MIL-STD-1815A, Department of Defense, 22 January 1983.

[Akers83] - Akers, Robert L., *A Gypsy-to-Ada Program Compiler,* Technical Report 30, Institute for Computing Science, The University of Texas at Austin, Austin, TX 78712, December 1983.

[Boyer80] - Boyer, Robert S., Moore, J Strother, *A Verification Condition Generator for Fortran,* Technical Report CSL-103, SRI International, June, 1980.

[DoD] - DeLauer, Richard D., "DoD Policy on Computer Programming Languages", Department of Defense Directive 5000.31, The Under Secretary of Defense, Washington, DC, 20301, June 1983.

[Good78] - Good, Donald I., Cohen, Richard M., Hoch, Charles G., Hunter, Lawrence W., Hare, Dwight F., *Report on the Language Gypsy, Version 2.0,* Technical Report ICSCA-CMP-10, Institute for Computing Science and Computer Applications, The University of Texas at Austin, Austin, TX 78712, September 1978.

[Luckham84] - Luckham, David C., von Henke, Friedrich W., Krieg-Brueckner, Bernd, Owe, Olaf, *Anna - A Language for Annotating Ada Programs, Preliminary Reference Manual,* Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

[Odyssey84] - Odyssey Research Associates, Inc. *A Verifiable Subset of Ada,* (Revised Preliminary Report), Odyssey Research Associates, Inc., 713 Clifton St., Ithaca, NY 14850, 14 September 1984.

Verifying Ada* Programs

Raymond J Hookway

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

December 14, 1984

The inductive assertion technique, which has been used successfully as a basis for verifying programs written in Pascal and some of its derivatives[9,10,15,17,20], is directly applicable to a large part of Ada. However, Ada also includes a number of constructs whose verification is not as well understood as the verification of constructs found in Pascal. These include packages, generic program units, tasks and exceptions The following is a description of our approach to the verification of these constructs

Packages

Packages in Ada can be used in a number of different ways. One way to use a package is just as the name of a collection of data and type declarations. This kind of use poses no special verification problems and can be handled using standard techniques.

The more important use of packages is their use for the implementation of abstract data types. Packages can be used to support data abstraction in two ways. One of these is to associate an abstract object with each package (or each instantization of a generic package). The entries to the package are then viewed as operations on the abstract object. This constitutes one of the "standard" approaches to data abstraction. Packages used in this way can be verified using the method first proposed by Hoare [11].

The other way to achieve data abstraction is to associate an abstract type with a private type which is declared in a package specification. Ada supports this kind of abstraction by restricting the operations that can be performed on a (limited) private type to (assignment and equality test plus) the entries to the package in which the type is declared. This latter approach to data abstraction is also found in Modula where types can be exported from a module. Packages which contain private types can be verified using techniques developed at Case (Ernst and Ogden [8], Hookway [14]) for the specification and verification of data abstraction in Modula programs.

Generic Program Units

Our approach to specifying and verifying generic program units is to allow generics to have parameters which are predicates and functions of the specification language. This is an extension of the usual Floyd Hoare assertion language found in the literature. A brief description of this approach is given below. A more detailed description is given in Ernst and Hookway [6].

Consider a generic program unit G that has a type parameter T and a procedure parameter p(x,y). (This description is not concerned with the types of parameters or Ada syntax.) The specifications of G depend on what p does. This can be specified by giving pre- and post-conditions for p. These assertions, like ordinary assertions, contain predicates and functions of the specification language (and also individual variables and constants). Unlike ordinary assertions, some of these predicates and functions are formal parameters of G just like T and p.

No special techniques are required to deal with type parameters (like T in the above example). This is because type parameters play the usual role of types in verification, they assure that the program is "well-formed". Specifications are also required to be well formed.

Each generic program unit is required to have a precondition which may contain specification language functions and predicates that are parameters to the generic. This assertion specifies the properties of these parameters which can be assumed in verifying the body of the generic.

*Ada is a registered trademark of the U.S. Department of Defense (OUSDRE-AJPO)

Instantiation of generic program units is handled by substituting actual parameters for formals The specifications of the resulting ordinary program unit (IG) are just those of the generic with formal parameters replaced by actuals This substitution removes all formal predicates and functions of the specification language The specifications of the instantiated program unit thus have the same form as a non-generic program unit Of course, it must be verified that the actuals satisfy the assumptions made about them in the generics specifications.

Tasks

We hope to adapt the method described in Ernst and Hookway [5], and Ernst [4] to the problem of verifying concurrent Ada programs. This method requires concurrent programs to be structured as a collection of modules** Each module defines one or more data abstractions, and any number of processes may be declared local to the body of the module. The purpose of these processes, called *realization processes*, is to manipulate the module's local variables in a way that does not affect the value of the abstract objects represented by the module. Although this is a very specific way to structure programs, it appears that most real software can be naturally structured in this manner. This approach allows modules to be verified separately even though the realization processes of one module execute concurrently with those of other modules.

Modules are verified by dividing the process and the entry procedures to the module into single mutex segments (SMSs) each of which contains at most a single critical section. The proof technique relies on the fact that, under certain restrictions, every concurrent execution of the SMSs produces the same result as some sequential execution of them. Sequential verification techniques can then be used to prove that the SMSs have the properties required for the module to meet its specifications.

The soundness of this approach depends on the fact that shared variables are accessed only under mutual exclusion. This is a severe restriction to place on the implementation. In order to ease this restriction, ownership specifications are added to modules. Ownership specifications allow shared variables to be treated as local to a process. Ownership is dynamic. A variable may be "owned" by one process at a given time and a different process at a later time. Processes are also allowed to "own" components of structured variables. Thus, one process can "own" one component of an array at the same time that another process "owns" a different component of the same array. However, it must be verified that two processes never "own" the same object at the same time and that processes only reference objects which they own.

The approach to verifying concurrent programs described above is the subject of active research at Case. Significant additional effort will be required to extend this approach to apply to Ada. In particular, the synchronization primitives used in Ada tasks are quite different from those studied by Ernst and Hookway [5] and exception handling in multi-task programs remains to be examined. Despite these difficulties, this appears to be a very promising approach to the verification of multi-task Ada programs.

Exceptions

Exceptions can be handled using the technique developed by Luckham [19]. Extensions to the this technique need to be developed to integrate exception handling with the techniques for data abstraction discussed above.

A Prototype Verifier

We are currently in the process of implementing a verifier for an Ada subset which is roughly equivalent to Modula. This implementation includes packages and private types. Addition of generics and exceptions, as described above, should be straight forward. The verifier will use the Case interactive theorem prover which is part of the Modula verifier described below.

**Modules correspond closely to packages in Ada and processes to tasks. The exact relationship of the concepts described in Ernst and Hookway [5] to Ada remains to be worked out.

A Design Environment

We feel that development of reliable software will require support of an integrated design environment. This environment should support a variety of approaches to verification from testing to theorem proving. However, it should be based on the notion of developing designs that are consistent with *precise* specifications. The environment should provide a framework for reasoning about designs. For example, it should track arguments about why portions of the design are correct, whether the arguments are based on test data, informal arguments, or formal (mechanical) proofs. Whatever form these arguments take, we expect them to be based on an understanding of what is required to formally verify the design.

We plan to build a series of incrementally updatable design environments based on the above ideas. The Ada verifier will be one component of these environments. Other components will include tools for developing and analyzing specifications, a facility for rapid prototyping, and a programming environment.

The Case Modula Verifier

The Case Verifier is an interactive system for verifying Modula programs. The verifier consists of two major components, a verification condition generator (vcg) and an interactive theorem prover. The source language is Modula, minus concurrent programming constructs and extended by the constructs described in Ernst and Ogden[8] and Hookway[14] for specifying Modula programs. The vcg generates verification conditions by symbolically executing the source program as described in Dannenberg and Ernst[3].

The theorem prover is an interactive, natural deduction theorem prover which was developed at Case. The design of this theorem prover is described in Ernst and Hookway[7]. The goal of this design was to produce a small, efficient theorem prover to support our research in verification methodology.

The verifier has been used to verify a small linking loader[14]. The loader is approximately four hundred lines long, divided equally between specifications and code. Selected verification conditions were proved using the theorem prover described above.

References

1. Ada Programming Language, Military Standard MIL-STD-1815, U.S. Department of Defense, December, 1980.

2. Formal Definition of the Ada Programming Language, Institut National de Recherche en Informatique et en Automatique, November, 1980.

3. Dannenberg, R.B. and Ernst, G.W., Formal Program Verification Using Symbolic Execution, *IEEE Transactions on Software Engineering*, Jan., 1982.

4. Ernst, G.W., A Method for Verifying Concurrent Processes, Report No. CES-82-1, Computer Engineering and Science Dept., Case Western Reserve Univ., 1982

5. Ernst, G.W. and Hookway, R.J., The Use of Data Abstraction in the Specification and Verification of Concurrent Programs, Internal Document, Computer Engineering and Science Dept., Case Western Reserve Univ., 1982

6. Ernst, G.W. and Hookway, R.J., Specification and Verification of Generic Program Units in Ada, Internal Document, Computer Engineering and Science Dept., Case Western Reserve Univ., 1982

7. Ernst, G.W. and Hookway, R.J., Mechanical Theorem Proving in the Case Verifier, *Machine Intelligence 10, Practice and Perspective*, Ellis Horwood Ltd., 1982, pp.123-145.

8. Ernst, G.W. and Ogden, W.F., Specification of Abstract Data Types in Modula, *ACM Transactions on Programming Languages and Systems*, Oct., 1980, pp.522-543.

9. German, S.M., An Extended Semantic Definition for Pascal for Proving the Absence of Common Runtime Errors, Report No. STAN-CS-80-811, Stanford University, June, 1980.

10 Good, D.I., Cohen, R.M., Hoch, C.G., Hunter, L.W. and Hare, D.F., Report on the Language Gypsy, Version 2.0, Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

11 Hoare,C.A.R., Proof of Correctness of Data Representations, *Acta Informatica*, 1972, pp.271-81.

12 Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *Communications of the ACM*, Oct., 1969, pp.576-580.

13. Hoare, C.A.R. and Wirth, N., An Axiomatic Definition of the Programming Language Pascal, Acta Informatica, 1973, pp.335-355.

14. Hookway, R.J., Verification of Abstract Data Types whose Representations Share Storage, Report CES-80-2, Computer Engineering and Science Dept., Case Western Reserve Univ., 1980.

15. Hookway, R.J. and Ernst, G.W., A Program Verification System, *Proc. of the Annual Conference of the ACM*, 1976, pp.504-508.

16. Ichbiah, J.D., Barnes, J.G.P, Heliard, J.C., Krieg-Brueckner, B., Roubine, O. and Wichmann, B.A., Rationale for the Design of the Ada Programming Language, *SIGPLAN Notices*, June, 1979.

17. Igarashi, S., London, R.L. and Luckham, D.C., Interactive Program Verification: A Logical System and Its Implementation, Acta Informatica, March 1975, pp.145-182.

18. London,R.L., Guttag,J.V., Horning,J.J., Lampson.B.W., Mitchell,J.G. and Popek,G.J., Proof Rules for the Programming Language Euclid, *Acta Informatica*, Jan. 1978, pp.1-26.

19. Luckham. D.C. and Polak, W., Ada Exception Handling - An Axiomatic Approach, *ACM Transactions on Programming Languages and Systems*, May, 1980.

20. Luckham. D.C., German, S.M., v.Henke, F.W., Karp, R.A., Milne, P.W., Oppen, D.C., Polak. W. and Scherlis. W.L., Stanford Pascal Verifier User Manual, Report STAN-CS-79-731, Computer Science Department, Stanford University, 1979

21. Luckham, D.C. and Suzuki, N., Verification of Array, Record and Pointer Operations in Pascal, *ACM Transactions on Programming Languages and Systems*, Oct. 1979, pp.226-244.

22. Wirth,N., Modula: A Language for Modular Multiprogramming, *Software--Practice and Experience*, Jan., 1977, pp.3-35.

Distribution List for IDA Paper P-1900

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John Solomond Director Ada Joint Program Office Room 3D139, The Pentagon Washington, D.C. 20301	2
Mr. John Faust Rome Air Development Center RADC/COTC Griffiss AFB, NY 13441	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. Richard Platek Odyssey Research Associates 1283 Trumansburg Rd. Ithaca, NY 14859-1313	5
IIT Research Institute 4550 Forbes Blvd., Suite 300 Lanham, MD 20706	1
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
IDA	
General W.Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Anne Douville, CSED	1
Ms. Audrey A. Hook, CSED	1
Dr. Richard J. Ivanetich, CSED	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. Terry Mayfield, CSED	1
Ms. Katydean Price, CSED	2
Mr. Steve R. Welke, CSED	1
Dr. Richard Wexelblat, CSED	1
IDA Control & Distribution Vault	3